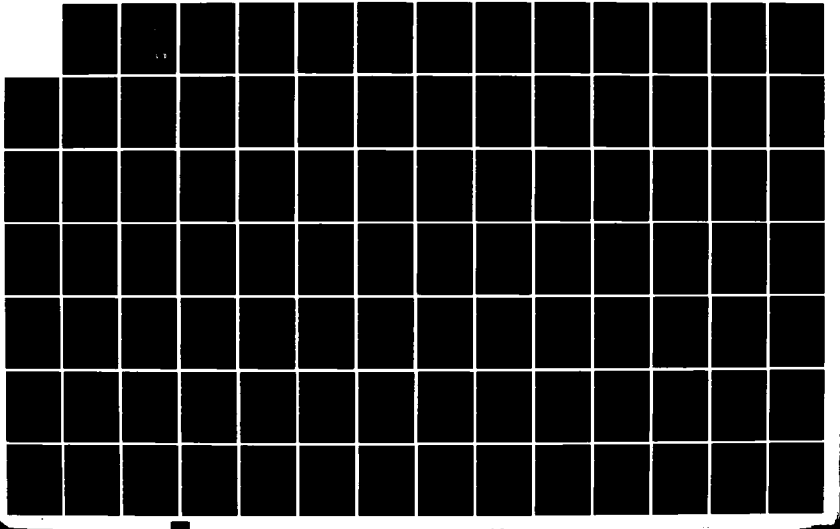
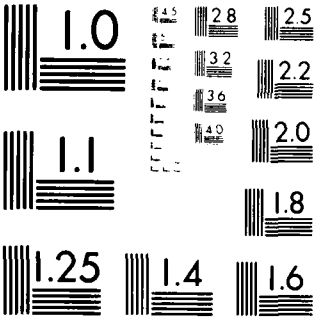
AD-A149 566 A CIRCUIT GRAMMAR FOR OPERATIONAL AMPLIFIER DESIGN(U) 1/1
MASSACHUSETTS INST OF TECH LEXINGTON LINCOLN LAB
A L RESSLER JAN 84 AI-TR-807 N00014-80-C-0622
UNCLASSIFIED F/G 9/2 NL





(2)

Technical Report 807

A Circuit Grammar For Operational Amplifier Design

Andrew Lewis Ressler

MIT Artificial Intelligence Laboratory

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AI-TR-807	2. GOVT ACCESSION NO. AD-A149 566	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) A Circuit Grammar For Operational Amplifier Design		5. TYPE OF REPORT & PERIOD COVERED Technical report 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Andrew Lewis Ressler		8. CONTRACT OR GRANT NUMBER(s) N00014-80-C-0622
9. PERFORMING ORGANIZATION NAME AND ADDRESS Artificial Intelligence Laboratory 545 Technology Square Cambridge, Massachusetts 02139		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Advanced Research Projects Agency 1400 Wilson Blvd Arlington, Virginia 22209		12. REPORT DATE January 1984 13. NUMBER OF PAGES 92
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Office of Naval Research Information Systems Arlington, Virginia 22217		15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Distribution of this document is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

Distribution is unlimited

18. SUPPLEMENTARY NOTES

Artificial Intelligence
Circuit
None

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Artificial Intelligence
Circuit
Computer-Aided Design
Design
Grammar
Language
Operational Amplifier

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Electrical circuit designers seldom create really new topologies or use old ones in a novel way. Most designs are known combinations of common configurations tailored for the particular problem at hand. In this thesis I show that much of the behavior of a designer engaged in such ordinary design can be modeled by a clearly defined computational mechanism executing a set of stylized rules. Each of my rules embodies a particular piece of the designer's knowledge.

(OVER)

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

Block 20 cont.

A circuit is represented as a hierarchy of abstract objects each of which is composed of other objects. The leaves of this tree represent the physical devices from which physical circuits are fabricated. By analogy with context-free languages, a class of ciucuits is generated by a phrase-structure grammar, of which each rule describes how one type of abstract object con be expanded into a combinacion of more concrete parts.

Circuits are designed by first postulating an abstract object which meets the particular design requirements. This object is then expanded into a concrete circuit by successive refinement using rules of my grammar. There are in general many rules which can be used to expand a given abstract component. Analysis must be done at each level of the expansion to constrain the search to a reasonable set. Thus the rules of my circuit grammar provide constraints which allow the approximate qualitative analysis of partially instantiated circuits. Later, more careful analysis in terms of more concrete components may lead to the rejection of a line of expansion which at first looked promising. I provide special failure rules to direct the repair in this case.

As part of this research I have developed a computer program CIROP, which implements my theory in the domain of operational design.

A Circuit Grammar

For Operational Amplifier Design

by

Andrew Lewis Ressler

S.B.C.S. S.B.E.E. Massachusetts Institute of Technology
1979
S.M. Massachusetts Institute of Technology
1981

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the
Requirements for the
Degree of

DOCTOR OF PHILOSOPHY

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

January 1984

Signature of Author Andrew L. Ressler
Department of Electrical Engineering and Computer Science
January 7, 1984

Certified by Gerald Jay Sussman
Associate Professor Gerald Jay Sussman
Department of Electrical Engineering
and Computer Science

Accepted by
Chairman Arthur C. Smith
Department of Electrical Engineering
and Computer Science

A Circuit Grammar
For Operational Amplifier Design

by
Andrew Lewis Ressler

Submitted to the Department of Electrical Engineering and
Computer Science on January x, 1984 in partial fulfillment
of the requirements for the Degree of Doctor of Philosophy in
Electrical Engineering and Computer Science.

ABSTRACT

Electrical circuit designers seldom create really new topologies or use old ones in a novel way. Most designs are known combinations of common configurations tailored for the particular problem at hand. In this thesis I show that much of the behavior of a designer engaged in such ordinary design can be modeled by a clearly defined computational mechanism executing a set of stylized rules. Each of my rules embodies a particular piece of the designer's knowledge.

A circuit is represented as a hierarchy of abstract objects, each of which is composed of other objects. The leaves of this tree represent the physical devices from which physical circuits are fabricated. By analogy with context-free languages, a class of circuits is generated by a phrase-structure grammar, of which each rule describes how one type of abstract object can be expanded into a combination of more concrete parts.

Circuits are designed by first postulating an abstract object which meets the particular design requirements. This object is then expanded into a concrete circuit by successive refinement using rules of my grammar. There are in general many rules which can be used to expand a given abstract component. Analysis must be done at each level of the expansion to constrain the search to a reasonable set. Thus the rules of my circuit grammar provide constraints which allow the approximate qualitative analysis of partially instantiated circuits. Later, more careful analysis in terms of more concrete components may lead to the rejection of a line of expansion which at first looked promising. I provide special failure rules to direct the repair in this case.

As part of this research I have developed a computer program, CIROP, which implements my theory in the domain of operational amplifier design.

This thesis describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research is provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0622.

To My Mother and Father

I would like to thank Gerry Sussman for superb supervision. He kept me on the right track; while encouraging me to learn more about physics, astronomy, electrical engineering and other areas outside of the realm of computer science.

I cannot begin to express my thanks and Love to my wife Linda and my daughter Julie for their moral support and trying to make me a "normal" person.

I would like to thank Patrick Winston and James Roberge for providing their expertise in artificial intelligence and electrical engineering, respectively.

Thanks to Brian Williams and Mike Komichak for their thought provoking discussions after reading many drafts of my thesis. Also, thanks to Elias Towe, Gerry Roylance, and Tom Knight for reading and commenting on drafts of my thesis.

Thanks to my sister Janet for "learning me" some English while editing several drafts of my thesis.

Thanks to Mike Kass for teaching me how to juggle and how to avoid becoming too serious.

CONTENTS

FIGURES

1. Introduction

Electrical circuit designers seldom create really new topologies or use old ones in a novel way. Most designs are known combinations of common configurations tailored for the particular problem at hand. Much of the behavior of a designer engaged in such ordinary design can be modeled by a clearly defined computational mechanism executing a set of stylized rules. The implementation of this model, CIROP, can design operational amplifiers such as the one in Figure 1. In this circuit, several commonly recognized collections of objects are circled, a differential pair, a current mirror, and the three main stages of the operational amplifier. Such a collection of objects can be considered a complex abstract object, made of internal components, and can be used as a component in a larger circuit. For example, Figure 2 shows typical representation of an operational amplifier as a circuit composed of three parts; the first, second and third stages. Each of these stages is a complex object composed of more primitive objects. A small set of such complex abstract objects can represent hierarchically a large number of valid circuits.

CIROP does suprisingly well at the task of circuit design. It designs circuits at the level of complexity described in Solomon's classic paper [1974]. Thus, CIROP demonstrates an upper bound on the amount of knowledge needed to achieve this level of competance.

1.1 Key Ingredients

Here are the key ingredients in the model of design that I present:

- Context-Free Hierarchical Expansion
- Failure Dependent Redesign
- Circuit Knowledge Formalization

Dividing a complex task into several smaller easier tasks is a well-known methodology for solving a problem [Simon]. Design begins with an abstract concept describing the ultimate object to be synthesized. The engineer proposes a method for building the object in terms of other known objects. This divides the problem into several smaller subproblems which are modeled naturally in a hierarchical fashion. The context-free assumption implies that the solution to each subproblem is independent of the other subproblems. Fortunately, the electrical engineering world is particularly amenable to this hierarchical paradigm of design, as exemplified by the operational amplifier. Its internal design can be characterized as a

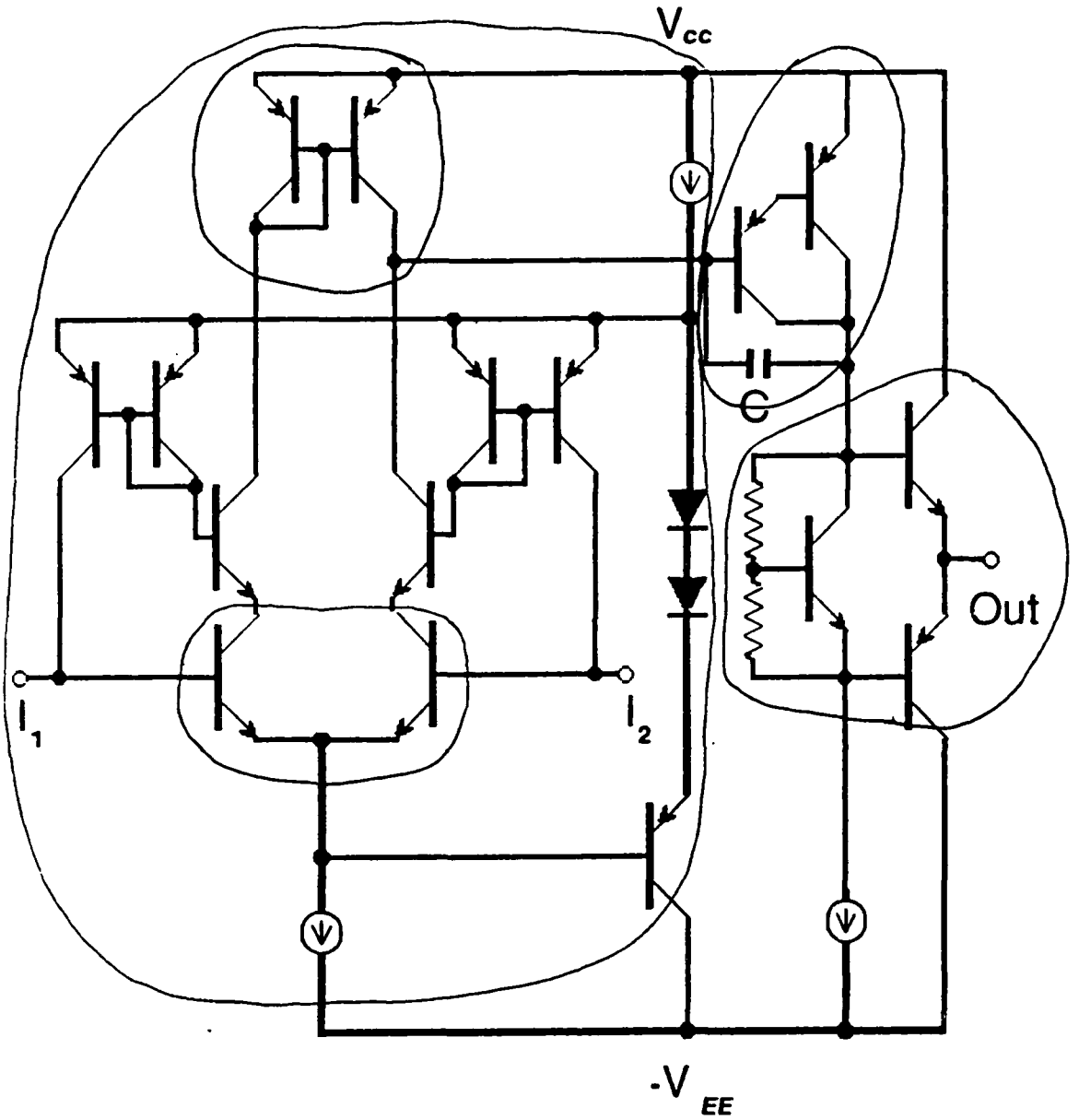


Fig. 1. Complex Operational Amplifier
Complex Amplifier that can be decomposed hierarchically.

---

hierarchy of objects, and the operational amplifier, itself, is usually a small piece in a larger design. The hierarchy is implemented as a phrase grammar, which is a set of rules that describe a space of possible circuit topologies. Any circuit generated by this grammar is valid. The grammar rules expand each abstract topological fragment (a non-terminal of the grammar) into a more concrete topological fragment. By

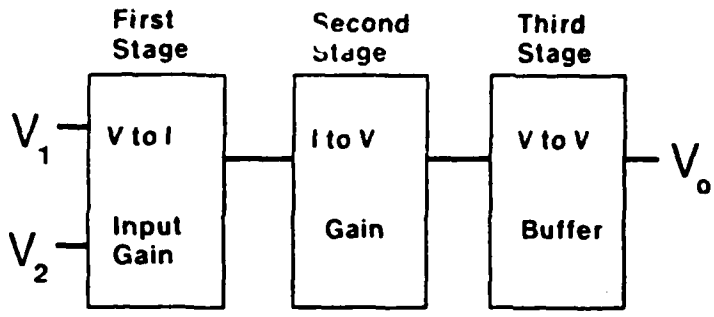


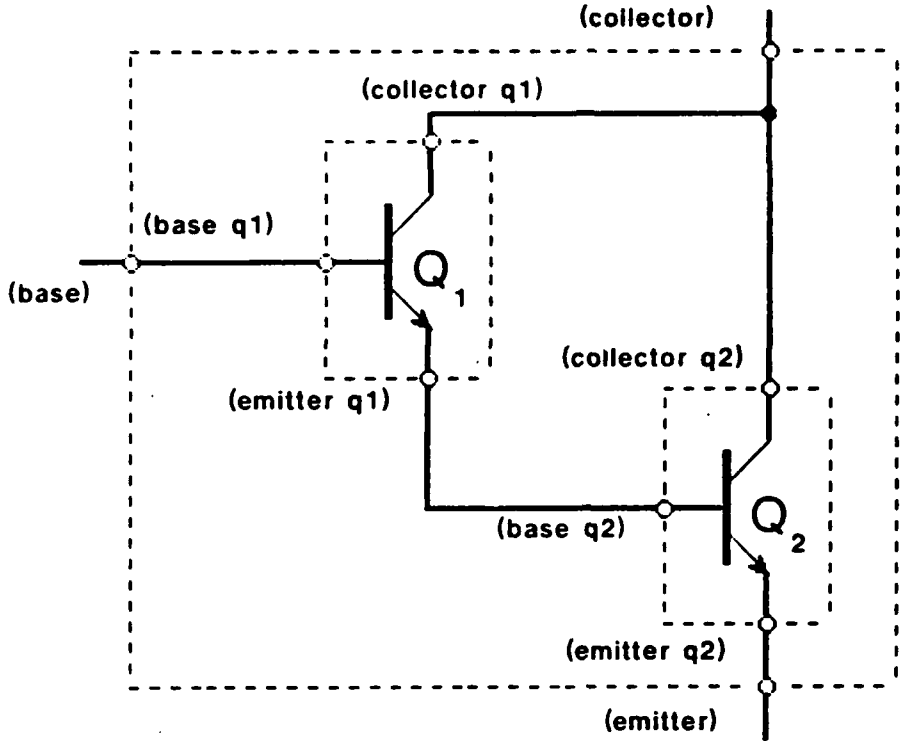
Fig. 2. Three Stage Operational Amplifier
Simple Representation of Three Stage Operational Amplifier

---

applying a sequence of these rules, one begins with an abstract topology and expands it until all the elements in the topology correspond to physical objects and need not be expanded further.

A typical simple abstract object is the darlington pair in Figure 3 composed of two internal transistors connected as shown. The equation in the figure is a constraint describing an aspect of the object's behavior and is used in the object's analysis.

There may be several ways to expand each abstract object in a hierarchy. Figure 4 describes two different rules in a circuit grammar that together recursively define a filter. The simplest filter is created by expanding the abstract filter goal using Rule 1. The next more complex filter is created using Rule 2 first. This expansion creates another goal of designing a filter which could be satisfied by using Rule 1. This example also shows that hierarchical descriptions of circuits applies to objects besides operational amplifiers.



β of darlington = β of Q1 * β of Q2
Fig. 3. Darlington Pair
A darlington pair is a virtual transistor that has higher β than a single physical transistor.

1.2 Formalizing the Domain

The concepts that must be formalized for design are as follows:

- Primitive Objects
- Composition Knowledge
- Analysis Knowledge
- Strategic Knowledge
  - Eliminating Candidate Circuits
  - Guiding Redesign of Circuits

As the physical basis of circuits, *primitive objects* are the vocabulary of the domain. Transistors,

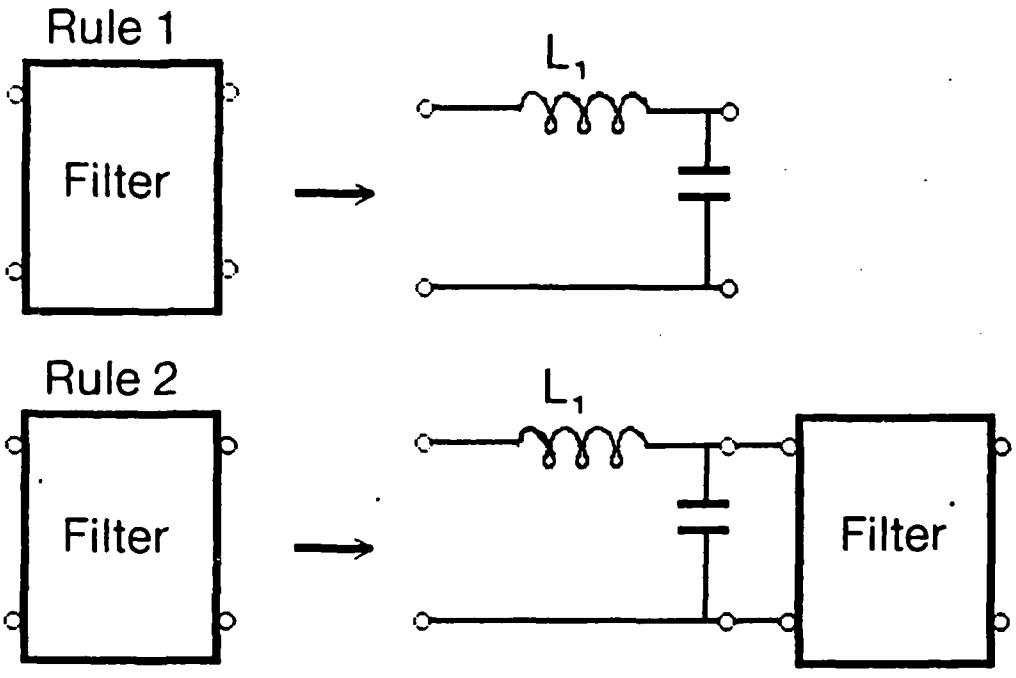


Fig. 4. Simple Recursive Circuit Grammar
The top part of the figure represents one way to build a simple filter. The bottom of the figure shows another way which is recursive.

---

resistors, and capacitors are typical primitive objects. *Composition knowledge* describes how to create complex abstract objects from the combination of other abstract and primitive objects. The composition knowledge is explicit in the phrase grammar rules. Each rule describes one way to combine objects to create an abstract object. *Analysis knowledge* verifies and tests the behavior of designed objects against the goals. *Strategic knowledge* controls the design process and includes the knowledge necesary to make the control decisions. Control is critical for both guiding the design towards a good proposal and guiding the redesign of an

unsatisfactory circuit.

1.3 CIROP

CIROP embodies vocabulary and composition knowledge in a hierarchical phrase grammar that models the hierarchical structure of circuits naturally. Each of its rules describes how to build a circuit object in terms of more basic objects. Analysis knowledge is represented as a set of constraints asserted with each rule that describes the behavior of the object and passes constraints to its component parts. Using these constraints, the circuit is analyzed and compared to the specifications given by the user. The constraints also represent strategic knowledge that restricts the search space of possible circuits. Failure rules formalize the strategic knowledge used by an engineer to correct the design of a circuit which fails to meet the specifications.

During expansion, strategic knowledge guides The design. Each rule proposed for expanding an object is checked for strategic knowledge about its applicability. If the strategic knowledge determines that the rule is not applicable, another rule is chosen to expand the current object. CIROP searches the space of possible circuits efficiently. When strategic knowledge shows that a rule is not applicable, CIROP avoids that entire region of the space.

The phrase grammar rules assert equations which describe constraints between the parts. These equations are used to analyze a rule's applicability and to verify a completed circuit against the specifications. If a specification is not met, CIROP determines which sections of the circuit do not meet specification. Then one of these sections is redesigned to improve the specification. Failure rules embody knowledge used to redesign circuits based on their failure. From the applicable failure rules, a rule is chosen. This failure rule suggests an alternative method of designing a particular piece of the circuit based on the specification that failed.

1.4 Control Flow in CIROP

The general procedure of CIROP is shown in Figure 5. The top level goal is a description of the abstract object to build and a set of specifications to meet. CIROP finds an appropriate rule to expand the abstract object and then analyzes the resulting circuit. If the circuit is not complete, CIROP goes back to the first step to expand some abstract object. The circuit is complete when all the abstract objects have been

expanded to physical objects. The complete circuit is compared with the original specifications. If the specifications are met, the circuit is finished. If not, a suggestion is made to improve the circuit. Suggestions describe a piece of the circuit to replace and an abstract object to replace the bad piece. If no suggestions are found, CIROP cannot design the specified circuit.

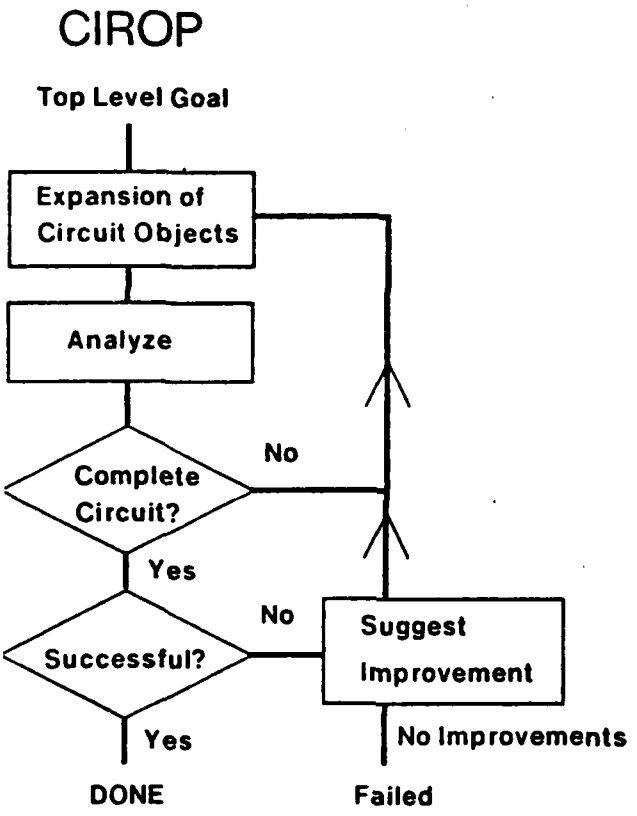


Fig. 5. Flow Chart of CIROP

1.5 Thesis Roadmap

Chapter 2 is a scenario of the design of a typical operational amplifier. It demonstrates that an operational amplifier can be described hierarchically and that specifications influence the creation of the circuit. Chapter 3 describes CIROP's phrase grammar rules in detail. Chapter 4 describes the algebra mechanism CIROP uses to analyze the circuits produced by its phrase grammar rules. Chapter 5 describes CIROP's control mechanism for guiding the design and redesign of the circuit. Chapter 6 details the scenario shown in Chapter 2. Chapter 7 discusses how CIROP relates to other research in the area of design. Chapter 8 summarizes the work and discusses limitations of this approach. Appendix 1 enumerates CIROP's phrase grammar rules. For general information on operational amplifiers, see the book by Roberge [1975].

2. Operational Amplifier Example

CIROP seems to do design in a human-like way because human designers can understand the knowledge it uses and the decisions it makes. The following scenario demonstrates a typical design by CIROP.

2.1 Scenario

The initial goal is to build an operational amplifier with the following specifications.

gain	500000.
input-offset-current	10 nanoamps
slew-rate	0.2 volts per micro-second
output-drive-current	15 milliamps
unity-gain-frequency	3 Megahertz
output-load-resistance	2 kohms
input-bias-current	0.2 microamps

Operational amplifiers have two voltage inputs and one voltage output. The main goal of an amplifier is to provide gain. The most common operational amplifier, shown in Figure 2, has three main stages. The first stage converts the differential input to single-ended and provides gain. The middle stage provides any necessary gain that the first stage cannot supply. The third stage buffers the output.

In general, CIROP has several choices for implementing each stage of the operational amplifier. Figure 6 is a partial tree of the hierarchy showing some options. For instance, the first stage can be implemented as a simple differential pair, a current cancellation differential pair, or a super beta differential pair. If the simple differential pair is chosen, it has two internal parts: the load and the emitter coupled pair. The load can be implemented as either a resistive load or a current mirror. In this design example, CIROP proposes the simple differential pair, shown in Figure 7, for the first stage, which is the simplest choice. Before proceeding with the design, the constraint between the gain and the input bias current is checked to see if the proposed first stage will be sufficient. Assuming that the second stage can provide lots of gain (maybe it will be a darlington pair) the first stage will need a transconductance (*gm*) of at least .0004 mhos. The

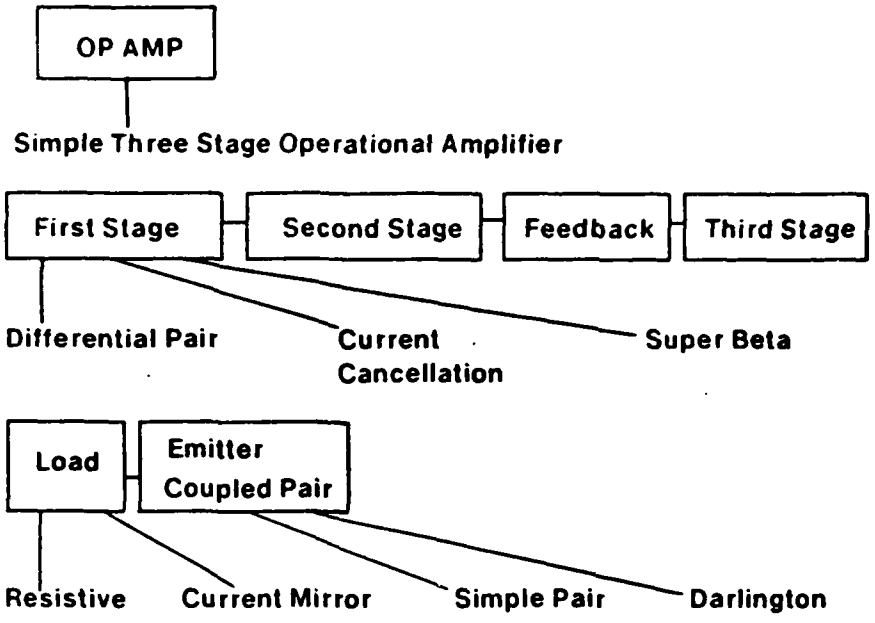


Fig. 6. Partial Tree of Operational Amplifier Hierarchy
Partial tree of the hierarchy implemented for designing operational amplifiers.

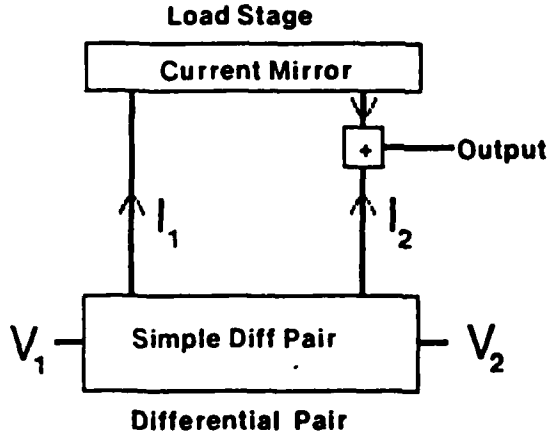


Fig. 7. Abstract First Stage
The first stage of an operational amplifier is usually composed of a differential pair with a load.

transconductance of a differential pair is related to the bias current as follows:[1]

$$gm = \frac{q\beta I_{bias}}{kT}$$

$$\text{Total gain} = gm * \text{Loading} * \text{Transresistance}$$

Therefore, the bias current will be at least 0.2 microamps, which meets the bias specification. The proposed first stage is adequate for now.

Figure 8 represents an abstract differential pair. It uses a matched pair of transistors for the emitter-coupled pair. A matched pair is a pair of transistors with nearly equal physical characteristics. Figure

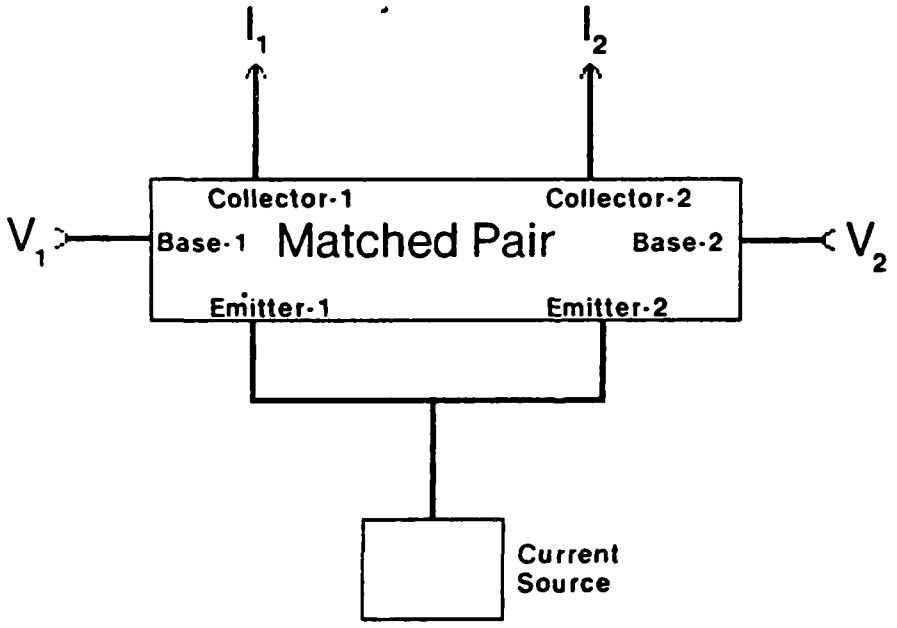


Fig. 8. Abstract Differential Pair
The simplest differential pair uses a matched pair and a load.

1. The values of the constants are as follows:
$\frac{kT}{q}$ = 0.025 Volts
β = 50 for pnp transistors
The β of a transistor is a property that depends largely on the device's physical characteristics.

9 is a matched pair.

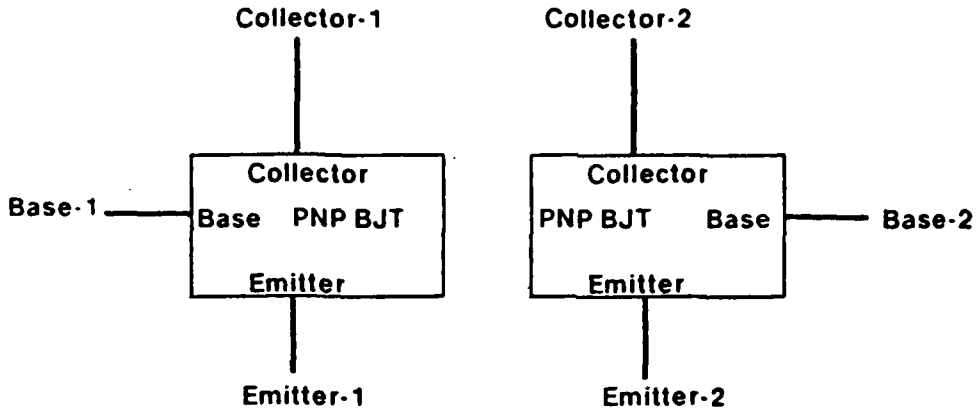


Fig. 9. Matched Pair
The matched pair is composed of two identical physical transistors.

Meanwhile, the second stage can be a simple common emitter stage, although a darlington pair may be necessary later. Continuing in this manner, the rest of the circuit is designed. The final circuit is similar to Figure 10.

However, a more careful analysis of the complete circuit reveals that the gain and the bias current specification cannot be met simultaneously. The first stage, the first stage matched pair, and the second stage transistor are the direct causes of the gain as shown in Figure 11.

To improve the gain CIROP must rebuild one of these options. CIROP's first choice is to replace the second stage's single transistor with a darlington pair. The improved circuit as shown in Figure 12 is analyzed and meets the specifications.

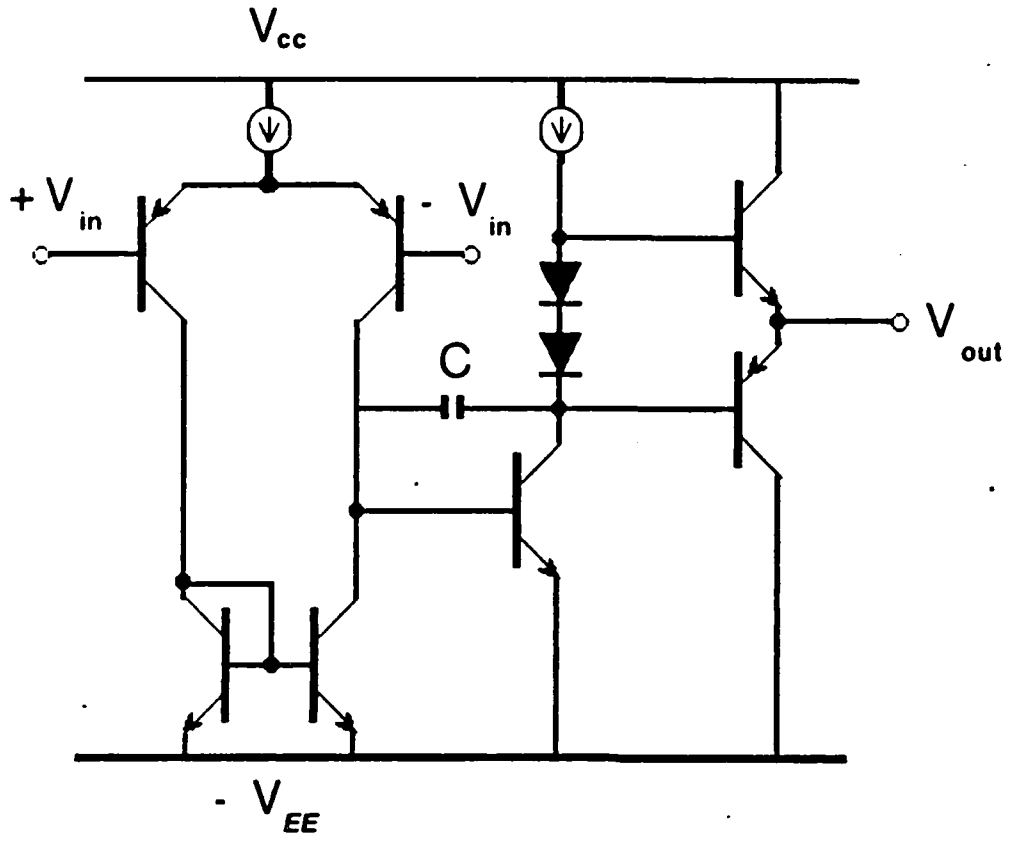


Fig. 10. Prototype Operational Amplifier
First Prototype for the operational amplifier being designed.

This example demonstrates CIROP using many of the types of design knowledge used by a human engineer. The vocabulary and composition knowledge are used to construct a valid circuit. Strategic knowledge is used to determine which parts are used to build the operational amplifier and how to rebuild an unsatisfactory prototype operational amplifier. Analysis knowledge is used to determine the performance of the circuit.

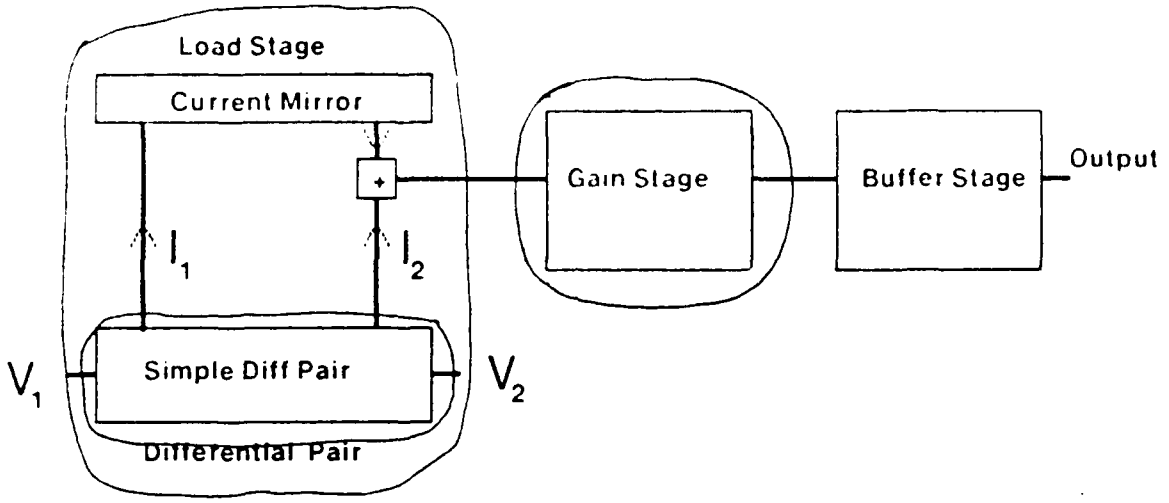


Fig. 11. Amplifier Stages to be Improved
The stages that might be redesigned to improve the gain are circled.

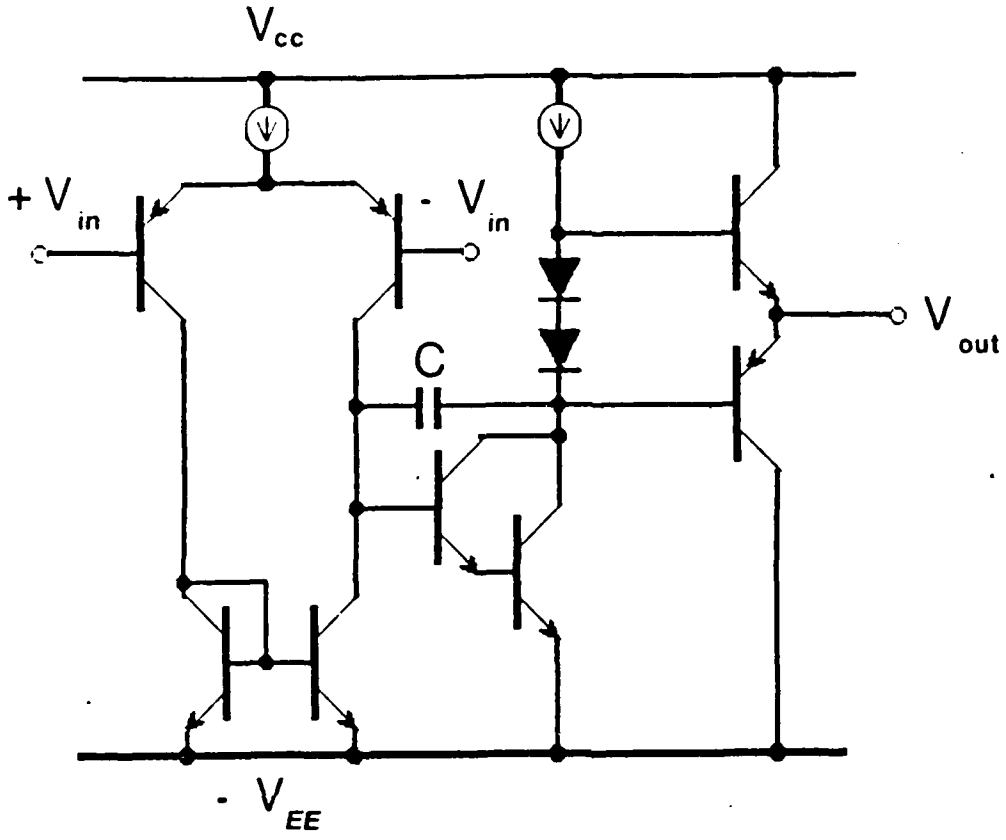


Fig. 12. Final Operational Amplifier
An amplifier designed by CIROP that required backtracking to improve the circuit by replacing a simple second stage with a darlington pair transistor.

3. Phrase Grammar

3.1 General Concept

A circuit phrase grammar is a set of rules specifying a circuit language by recursive expansion. The rules show how a (non-terminal) symbol may be expanded into a combination of symbols. This expansion process is repeated until no non-terminals are left (objects which could be expanded).[1] The terminal elements in the grammar are physical objects, and the resulting topology is a legal sentence of the grammar. The grammar describes a set of topologies that perform useful functions. The grammar is extensible since new rules can be added at any time.

In contrast to a language grammar, a circuit grammar specifies how to combine components to make compound structures. In language grammars, the only method of combining parts in surface language is juxtaposition, whereas in circuits one must specify the resulting topology. Figure 13 is an example of a simple language grammar.

Grammar Rule
S :: aS | a

Example derivations of sentences of above grammar where S is the top level object:
S =>a
S =>aS =>aa

Fig. 13. Context Free Language Grammar Example

CIROP's phrase grammar rules have two main parts: the pattern and the body. The pattern describes the main characteristics of the abstract object that it builds and determines if the rule is applicable when building an object. The body is a list of assertions describing how the object is built and analyzed. There are three types of assertions used. The new-part assertions describe internal components. The connection

1. For more information on language grammars see Aho[1979].

assertions create the topology by connecting the internal components to each other and to the abstract object. The constraint-equation assertions create constraints between the components and the abstract object. CIROP's rules include the relations between the internal objects, that can be used to simplify analysis of the resulting circuit.

```
;; Typical structure of a phrase grammar rule.
(to-make-a name
           pattern
           new-parts
           connections
           constraint-equations)
```

3.2 Phrase Grammar Pattern

Every phrase grammar rule includes a pattern that is used to determine if the rule can create the currently proposed object. The pattern includes the following three components: the *type* property, the *has* properties, and the *specification* tradeoffs. The pattern for the simple differential pair is as follows:

```
(where (type amplifier)
       (has (input differential))
       (has (input voltage))
       (has (output single-ended))
       (has (output current))
       (has (sign ?sign))
       (has (simplest 1))
       (with-specs
        (= transconductance (* bias-current beta q/kT))
        (= offset-voltage simple-offset-voltage)))
```

The *type* property says that the differential pair is an amplifier. The *has* properties list a variety of properties that a differential pair will have. The *simplest* property of 1 means that this is the simplest method of building this type of object. The *with-specs* denotes a list of specification tradeoffs. In this example, the transconductance is equal to the product of the bias-current and the constants *beta* and *q/kT*. This is the tradeoff used in the example in Chapter 2 to check if the rule was applicable.

A pattern may include variables which are denoted with a "?" as is the *?sign* variable in the example. When the pattern is matched, the variable will match anything in the corresponding position in the matching pattern. This binds the value of the variable so that any place in the rule that uses the variable will have that value. In the example pattern the *?sign* variable is used to pass the type of transistor to use inside the

differential pair.

3.3 New Parts

NEW-PART is the assertion that specifies a component of the rule currently being expanded. The assertion's three arguments are the name, pattern, and use. The part name allows equation assertions in the rule body to reference properties of the object. The pattern is used to match rules applicable to creating it. When matching a rule's pattern, the *type* property must match exactly, and every *has* property in the NEW-PART pattern must be in the rule's pattern. *Has* properties in the rule's pattern are not required to be in the NEW-PART's pattern. Each NEW-PART may also have a *use* property, which is examined in the analysis phase of CIROP. Following is an example of a NEW-PART assertion used in the rule for a three-stage amplifier to specify a differential stage for its first stage. This NEW-PART assertion matches the example used to describe rule patterns in section 3.2. In this example the *sign* property is *pnp*, so that the value of the *?sign* variable in the rule pattern from section 3.2 will be *pnp*.

```
(new-part first-stage
          ((type amplifier)
           (has (input differential))
           (has (input voltage))
           (has (output single-ended))
           (has (output current))
           (has (sign pnp)))
          (use ((gain gm) (slew-rate gm))))
```

The process of finding a rule to expand a NEW-PART is more than simple pattern matching. It is a negotiation between the NEW-PART's pattern, which is an advertisement for a part, and a set of rules which are the applicants for filling the job of that part. CIROP mediates between the applicants. First, CIROP considers only the applicants of the right type. CIROP eliminates any that do not have the specialties (the *has* properties) required for the particular part so that the remaining candidates have the right properties. So far all CIROP has done is simple pattern matching. Now it must determine the best candidate. Assuming that the simplest is best, CIROP sorts the candidates, simplest to most complex. Finally, it asks each candidate, starting with the simplest, if it thinks it can do the job. The first candidate that replies "yes" is chosen for the job. A candidate answers "no" if its specification tradeoffs cannot possibly meet the specifications. Since

CIROP can backtrack, it is allowable for the candidate to reply "yes" even if at the end it is not sufficient.

3.4 Connections

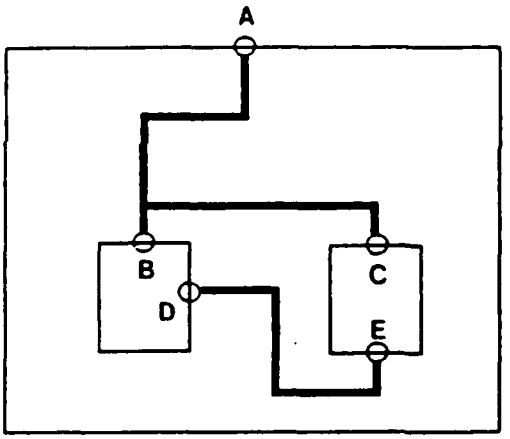
Terminals of objects are connected using one of the assertions **CONNECT**, or **RAIL-CONNECT**. These assertions are the glue that provides the topological structure to the circuit. The **CONNECT** assertion states that a group of object's terminals are connected together. This assertion automatically generates an assertion corresponding to Kirchoff's Current Law (KCL), which states that the sum of the currents flowing into the mentioned terminals equals zero.

The current's direction is positive for current flowing into an object's terminal. If a connection includes terminals from an abstract object to one of its component terminals, it negates the current flow from the abstract terminal when creating the KCL[1] assertion. This is shown in Figure 14.

The **RAIL-CONNECT** assertion states that the mentioned terminals are connected to either the power or ground bus of the circuit. No KCL assertions are generated from the **RAIL-CONNECT** assertions since it would violate the hierarchy for one rule to connect all the bus terminals together.

---

1 The examples in this thesis do not explicitly mention Kirchoff's Voltage Law (KVL), which states that the sum of the voltages around a closed loop is zero. This is not indicative of a limitation in the theory, but that the particular examples did not need it for their analysis. Most of the internal operation of an operational amplifier can be analyzed in terms of current flows with little reference to voltage loops



(Connect D E)

0 = (Current D) + (Current E)

(Connect A B C)

0 = −(Current A) + (Current B) + (Current C)

Fig. 14. KCL Assertions
CIROP creates KCL assertions from the CONNECT assertions as shown. *The current from terminal A is* negated since it is from the surrounding abstract object.

3.5 Equations

The knowledge used for analysis is contained in the equation assertions, which assert relations between properties of the parts, or propagate constraints from an abstract object to its internal parts. To preserve the creation rule's hierarchical nature, the equations must refer only to the abstract object's properties or any part's advertised properties. An advertised property is a property of a part included in all methods for building that part. For instance, if a virtual transistor is part of a circuit, the object requiring the virtual transistor does not know whether the virtual transistor will be implemented as a single transistor or as a darlington pair. Therefore, the object cannot refer to a parameter such as the β of the first transistor in a darlington pair implementation of the virtual transistor, but only to the β of the combination.

When CIROP solves these equations, an equation might need to know which variables to leave as

unknowns, and which variables to solve for, immediately. For instance, solving for the β of a transistor, in terms of the *gm* and the collector current, is useless, since β is a physical constant. To solve this problem, CIROP uses "variable priority."[1] That is, in solving an equation, the system examines the priority of each variable that it could solve for, and then solves for the highest priority variable. When variables have the same priority, CIROP can choose any one. In general, a priority depends on the level of abstraction of the associated object. An abstract object's variable has higher priority than a less abstract object's variable.

Sometimes, CIROP should solve for a specific variable, regardless of priorities. In such a situation, the rule specifies which variable should be solved. A typical example is as follows, where the variable **(transresistance second-stage)** has the highest priority.

```
(equation-with-variable-priority
  (= (transresistance second-stage)
     (* (current-gain second-stage)
        (input-resistance third-stage)))
  (transresistance second-stage))
```

The equations used in the phrase grammar rules are similar to ones human designers use. It is natural to express such equations in the form V = f(A, B, C), where V is an abstract variable, and A, B, and C are more concrete variables.[2] It is natural to solve equations in terms of the variable associated with most abstract object. For instance, the β of a darlington pair is the product of the βs of the individual transistors. The equation $\beta = \beta_1 * \beta_2$ where β is the darlington's β and β_1 and β_2 are individual transistors' βs is natural; the equation $\beta_1 = \beta / \beta_2$ is not. Empirical results show that using variable priority improves the performance of the algebra system. Without variable priority the following problem often appears. The algebra system cannot factor the numerator of the following expression and cannot reduce it to "c + d".

$$\frac{ac + ad + bc + bd}{a + b}$$

---

1. A simple form of variable priority is used in CIRCOM, a system developed for teaching electrical circuits in a computational context by Gerry Sussman.

2. The most concrete variables are parameters associated with physical objects. The most abstract variables are specifications associated with the top level abstract object in a design.

3.6 Interpreting Phrase Grammar Rules

The creation of a circuit starts with a queue of one NEW-PART assertion, which describes the high-level abstract object to be designed. The procedure for expanding the rules is:

Repeat the following steps until the queue is empty.

1. Take the next assertion off the queue of NEW-PART assertions.
2. Find the rules that match the type in the pattern of the new part. This is done by simple lookup since the rules are indexed by the type property.
3. Of the rules found in step 2, eliminate those without the required has properties mentioned in the NEW-PART's pattern.
4. From the remaining rules, find the simplest rule that meets the specifications.
5. Expand the rule. Each assertion in the rule-body is sequentially asserted, and depending on the assertion type, an appropriate action is taken. Any NEW-PART assertions in the rule's body are added to the end of the queue; this causes the design space to be searched in breadth first order.

For example, when expanding a NEW-PART assertion for a virtual-bjt-transistor[1], two rules are found in step 2: one for a single-transistor-bjt and one for a double-darlington-transistor-bjt. The rule names and their associated patterns are as follows:

```
Single-transistor-bjt
(where (type virtual-bjt-transistor)
       (has  (sign ?sign))
       (has  (simplest 1)))

Double-darlington-transistor-bjt
(where (type virtual-bjt-transistor)
       (has  (sign ?sign))
       (has  (simplest 2)))
```

In step 3, neither of the rules can be eliminated since they have the same sign has property. In step 4, the single-transistor-bjt is chosen since it has a lower simplest number. Then in step 5, the chosen rule is expanded. In this example, the virtual transistor creates a single real transistor.

---

1 A BJT is a bipolar junction transistor.

3.7 Phrase Grammar Rule Examples

The following subsections describe some of CIROP's typical grammar rules.[1]

3.7.1 Physical Transistor Rule

A physical transistor is a terminal node in the phrase structure grammar. It corresponds to a real transistor and has properties, such as β, that depend on the transistor's physical characteristics. The transistor is the main component in an integrated operational amplifier, since it can be used for applications other than the normal transistor application, such as that of a diode or resistor. Here is the grammar rule that describes an npn BJT. The rule includes equations used to model the standard BJT.

```
(to-make-a simple-npn-bjt
   ;; Pattern to match:
1  (where (type bjt)
          (sign npn)
2         (has (simplest 1)))
3  (equation-with-variable-priority
    (= (current (collector)) (* (beta) (current (base))))
    (current (collector)))
   (= (beta) npn-beta)
4  (= (gm) (* q/kT (current (collector))))
   (= (rpi)(/ (beta) (* q/kT (current (collector)))))
5  (equation-with-variable-priority
    (= (ro)(/ 200. (current (collector))))
    (ro))
6  (= 0 (+ (current (collector)) (current (base)) (current (emitter)))))
```

Simple-npn-bjt is the rule's name. Line 1 starts the pattern describing the type of object that the rule will synthesize. Line 2 says that this rule will create the simplest npn BJT. Lines 3 and 5 are examples of a variable simplest assertion wrapped around an equation. Line 4 is the familiar relation between the *gm* and the transistor's collector current. Line 6 states the KCL relation for the transistor's terminals. This rule includes no NEW-PART or CONNECT assertions since it is a primitive circuit element.

---

1. For readability, the notation of the rules is modified slightly from the actual form used by CIROP. Appendix 1 contains the correct forms for all rules used in CIROP.

3.7.2 Virtual Transistor Rule

A virtual transistor is any circuit that approximates a physical transistor's behavior. This object is commonly expanded into a single physical transistor, the simplest type of virtual transistor. When a higher β is required than can be produced by a single transistor implementation, the designer may choose to use a darlington pair which is another virtual transistor. This is composed of two physical transistors connected as shown in Figure 3. The rule that creates the darlington is:

```
(to make-a double-darlington-transistor-bjt
   ;; Pattern to Match:
   (where (type virtual-bjt-transistor)
1         (has  (sign ?sign)))
   ;; New Parts
2  (new-part q1 '(type bjt)(has (sign ?sign))))
3  (new-part q2 ((type bjt)(has (sign ?sign))))
   ;; Connections
4  (connect (base)(base q1))
   (connect (collector)(collector q1)(collector q2))
   (connect (emitter)(emitter q2))
   (connect (emitter q1) (base q2))
   ;; Equations governing analysis
   (= (rpi)  (* 2 (beta q1) (rpi q2)))
   (= (ro)   (ro q2))
5  (= (beta) (* (beta q1) (beta q2)))
   (= (gm)   (* (/ 1 2)  (gm q2))))
```

Lines 2 and 3 assert the two transistors that are used to build the darlington. Line 4 is an example of the CONNECT assertion, which connects the base of the darlington to the base of the internal transistor q1. Line 5 is the familiar relation where the β of a darlington pair is the product of the internal transistors' βs.

In Line 1, the symbol *?sign* is a variable with the name *sign*. When matching the rule's pattern against the assertion's pattern, a value of either npn or pnp should be matched against this variable. Since the body of the rule is expanded within the environment of the pattern match, the value found for *?sign* is used throughout the rule. Thus the sign of the darlington is passed to the physical transistors created in lines 2 and 3.

3.7.3 Three Stage Operational Amplifier Rule

The following rule is the top level rule used in CIROP to describe the simplest standard way to build an operational amplifier. It creates the three main stages and the feedback stage for the operational amplifier. The constraint equations pass the input specifications down to these stages.

```
(to-make-a three stage-operational-amplifier
   ;; Pattern to match:
   (where (type amplifier)
          (has (input voltage)(input differential)
               (output voltage)(output single-ended)(simplest 1)))
   ;; Parts are:
1  (New-part first-stage
             ((type amplifier)
              (has (input differential)(output single-ended)
                   (input voltage)(output current)(sign pnp)))
             (use ((gain gm) (slew-rate gm))))
   (new-part feedback ((type capacitor)))
   (new-part second-stage
             ((type amplifier)
              (has (input single-ended)(output single-ended)
                   (input current)(output voltage)(sign npn))))
   (new-part third-stage
             ((type buffer)(has (input voltage)(output voltage))))
;; Connections to outside are:
2  (connect (t+) (t+ first-stage))
   (connect (t-) (t- first-stage))
   (connect (ot) (ot third-stage))
;; Internal connections are:
3  (connect (ot first-stage)(it second-stage)(t1 feedback))
   (connect (ot second-stage)(it third-stage)(t2 feedback))
;; Propagating specifications to parts:
   (= (power-consumption)
      (+ (power-consumption first-stage)
         (power-consumption second-stage)(power-consumption third-stage)))
4  (= (slew-rate)(// (max (current (ot first-stage)))(capacitance feedback)))
   (= (cutoff-frequency)(// (transconductance first-stage)(capacitance feedback)))
   (= (offset-current) (offset-current first-stage))
5  (= (input-bias-current)   (input-bias-current first-stage))
   (= (input-offset-voltage) (input-offset-voltage first-stage))
   (= (drive-current)(maximum (current (ot third-stage))))
   (= (gain) (* (transconductance first-stage)(loading second-first-stage)
                (transresistance second-stage)))
   (equation-with-variable-priority
    (= (transresistance second-stage)
       (* (current-gain second-stage)(input-resistance third-stage)))
    (transresistance second-stage))
   (equation-with-variable-priority
6   (= (loading second-first-stage)
      (// (output-resistance first-stage)
          (+ (output-resistance first-stage)(input-resistance second-stage))))
    (loading second-stage first-stage))
   (equation-with-variable-priority
    (= (load-resistance second-stage)(input-resistance third-stage))
    (load-resistance second-stage))
   (= (distortion) (distortion third-stage))
   (= (voltage-gain third-stage) 1))
```

Line 1 is the **NEW-PART** assertion for creating the first-stage which will be a pnp differential pair with a load. The *use* properties show that the stage has a major affect on the gain and the slew-rate. If the resulting circuit fails and the tradeoff involves one of these specifications, then this part will be a candidate for replacement with an appropriate failure-control-rule. Line 2 connects the t+ input terminal of the operational amplifier to the t+ input terminal of the first stage differential pair. Line 3 connects the output terminal of the first-stage to the input terminal of the second stage and t1 terminal of the feedback stage. Line

4 asserts that the slew-rate is the quotient of the maximum current flowing out of the first-stage and the capacitance of the feedback stage. This assertion is dependent on several implicit assumptions. First, the slew-rate is limited by the charging of the feedback stage's capacitor by the current available from the first stage. Second, it is assumed that all of the current flowing from the first stage is available to charge the capacitor. Line 5 makes the assumption that the input bias current is directly dependent on the first stage, so the constraint is passed unchanged to the first stage. Line 6 shows the relation of the loading on the output resistance of the first stage and the input resistance of the second stage. The loading is a necessary component of the gain relation of the operational amplifier since it acts as a current divider and decreases the available gain.

4. Analyzing the Circuit

4.1 Hierarchical Equations

Each rule for creating an object asserts equations that are used to analyze the object. These equations state constraints between internal parameters and external parameters. For instance, the rule for building a physical transistor includes the device law that relates the *gm* of the transistor to the current flowing into the collector terminal. The equations associated with physical devices represent commonly used laws of network theory such as KVL, KCL, and device laws. The term "Hierarchical Equations" is used because abstract objects can also have equations to represent the behavior and analysis of the object. Thus, a hierarchy of equations exists constraining things from the highest level object to the lowest level object. The given specifications influence circuit design since they are referenced in the equations of the highest level object.

High level equations greatly aid the analysis process. For example, the top level rule of a typical operational amplifier has the following list of equations describing the amplifier's gain.

```
gain = (transconductance first-stage)
       * (loading second-first)
       * (transresistance second-stage)

(transresistance second-stage) = (current-gain second-stage)
                                 * (input-resistance third-stage)

(loading second-first) =
                          (output-resistance first-stage)
       ---------------------------------------------------------------
       (output-resistance first-stage) + (input-resistance second-stage)
```

The variable *gain* is the name of the specification variable for the operational amplifier's gain. The equations state that the gain is dependent on the following variables:

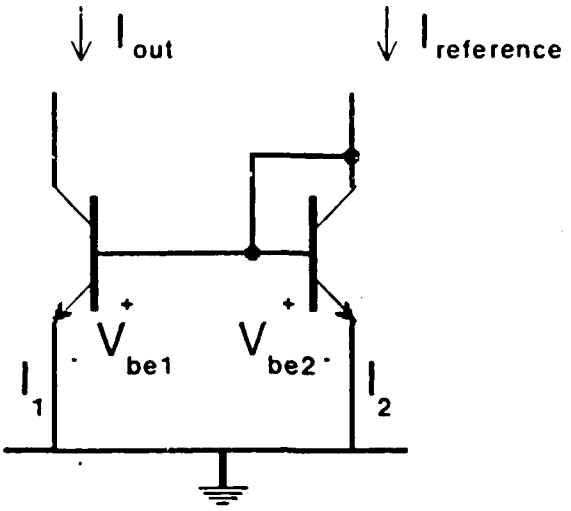
```
(transconductance first-stage)
(current-gain second-stage)
(output-resistance first-stage)
(input-resistance second-stage)
(input-resistance third-stage)
```

High-level equations can ignore details that are irrelevant to a variable's solution, if including them would complicate matters unnecessarily. Note that the operational amplifier's gain is not dependent on the third stage's gain, which is usually close to one and thus affects the overall gain very little. By ignoring the third stage's gain, the calculation of the overall gain can ignore possibly tedious calculations for the gain of the

third stage. Of course, the third stage is then constrained to be a buffer stage with gain of approximately "1".

Another advantage of hierarchical equations is the ability to bypass calculations too complex to solve by presolving some of the equations or using approximations [Roylance75]. Devices such as transistors have complex equations involving exponentials to describe their complete behavior. The current mirror depicted in Figure 15 relies on symmetry to perform the goal of providing nearly equal currents in the collectors, but cannot be analyzed without using the exponential relationship of the emitter current to the base-emitter voltage. Fortunately, using an equation as part of the abstract current mirror's grammar rule, the currents in the emitters can be stated to be identical, which provides the required constraint while bypassing the exponentials.

Hierarchical equations provide other advantages. Analysis results are more easily understood since the high level equations provide structure to the answers. If analysis were done at only the KVL and KCL level, it would be difficult to track down the reason behind a particular result.



Goal:

$I_{out} = I_{reference}$

Complex equations:

$V_{be1} = V_{be2}$

$I_1 = I_s(\exp(\frac{qV_{be1}}{kt}) - 1)$

$I_2 = I_s(\exp(\frac{qV_{be2}}{kt}) - 1)$

Presolved Result used in Analysis:

$I_1 = I_2$

Fig. 15. Current Mirror

The current mirror's behavior is determined by a complex relationship involving exponentials. Hierarchical equations allow CIROP to bypass calculations involving exponentials by using results that have been precomputed by the writer of the rules.

4.2 ORACLE - An Incremental Algebra System

The ORACLE incremental algebra system is used to solve the equations in the analysis of the circuits. As each equation is presented to the algebra system one of the unknown variables in the equation is solved for symbolically. This value and information about the equations it depends on are retained in the ORACLE database. The equation's assertion retains the name of the variable that was solved. The ORACLE is incremental because it is given equations one at a time and solves the equation based only on the results stored in a database of previously solved values.

The algebra system is a Truth Maintenance System [Doyle] for the solving of algebraic equations. The algebra system can selectively undo the effects of equations that are no longer considered true while retaining information derived from the equations that are still considered true. Because of this when rebuilding a circuit part, CIROP only needs to re-analyze the affected equations. The ORACLE is similar to other systems, such as EL [Stallman & Sussman], that use propagation of constraints to solve algebra and use truth maintenance to retract assertions. It's major difference is that it separates the algebra and the truth maintenance from the system that uses it. This provides a clean interface so that implementation issues of CIROP are not affected by the implementation of the algebra solving mechanism.

4.2.1 Sample

The following example demonstrates the capabilities of the ORACLE to solve equations and undo the effects of equations selectively. The following set of equations are presented to the ORACLE in the order shown.

1. C = A + B
2. E = B + C
3. G = D + F
4. H = C + D
5. M = D - K

Solve equation 1 for the result[1] that "C = A + B." Solve equation 2; however, instead of solving the equation "B + C = E", substitute the known value of C from equation 1 so that equation 2 becomes "A + 2B = E." The solution of this equation is "B = (E - A)/2." Here is a table of results so far.

---

1 To demonstrate how ORACLE works, it does not matter which variable is solved for. When analyzing circuits, the variable solved for depends on the variable's priority. In this example the variable that is solved for is underlined.

	Equation	Equation Solved	Variable-Value
1	C = A + B	C = A + B	C : A + B
2	E = B + C	E = A + 2B	B : (E - A)/2

The items in the column labeled *Equation* are the original equations. The items in the column labeled *Equation Solved* are the equations after substituting the values of known variables. The items in the *Variable Value* column are the solutions of the equations. The complete table is as follows:

	Equation	Equation Solved	Variable-Value
1	C = A + B	C = A + B	C : A + B
2	E = B + C	E = A + 2B	B : (E - A)/2
3	G = D + F	G = D + F	G : D + F
4	H = C + D	H = (A + E)/2 + D	A : 2H - 2D - E
5	M = D - K	M = D - K	D : K + M

Whenever the ORACLE is asked for a variable's value, it returns an expression that contains no variables with a known value. Since equation 5 has solved for the variable D, if one asked for the value of G, the result would be "G : F + K + M." Figure 16 shows a dependency network of the equations contained in the above table. The arrows point from known results to equations dependent on those results.

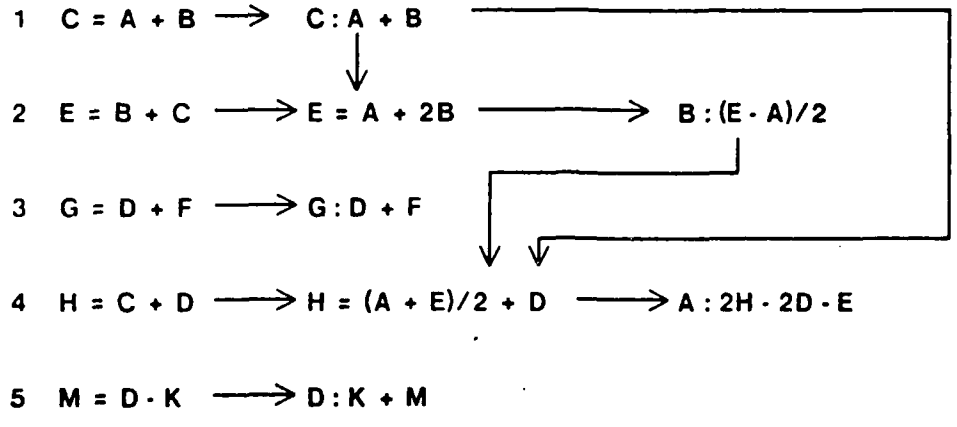


Fig. 16. Algebra Example
This shows the dependencies of the equations solved.

The interesting part comes when an equation is retracted. Suppose that equation 2 is a consequence of a bad design decision and is retracted. Obviously, that affects the value calculated for the variable B since equation 2 was solved for the variable B. It also affects the value of variable A calculated in equation 4. Before equation 4 was solved it was transformed using a substitution that depended on the result of equation 2. After retracting equation 2 we get the following.

	Equation	Equation Solved	Variable-Value	Equation-Truth	Value-Truth
1	C = A + B	C = A + B	C : A + B	True	True
2	E = B + C	E = A + 2B	B : (E - A)/2	False	False
3	G = D + F	G = D + F	G : D + F	True	True
4	H = C + D	H = (A + E)/2 + D	A : 2H - 2D - E	True	False
5	M = D - K	M = D - K	D : K + M	True	True

The new column *Equation-Truth* is the truth value of the equation. Since equation 2 was retracted, its truth value is false while the other equations are still true. The new column *Truth-Value* is the truth value of the value solved for with the equation. The value resulting from equation 2 is false; therefore the value resulting from equation 4 is false also, although equation 4 is still true. Figure 17 shows the same equations as the Figure 16, with the false information marked with large crosses.

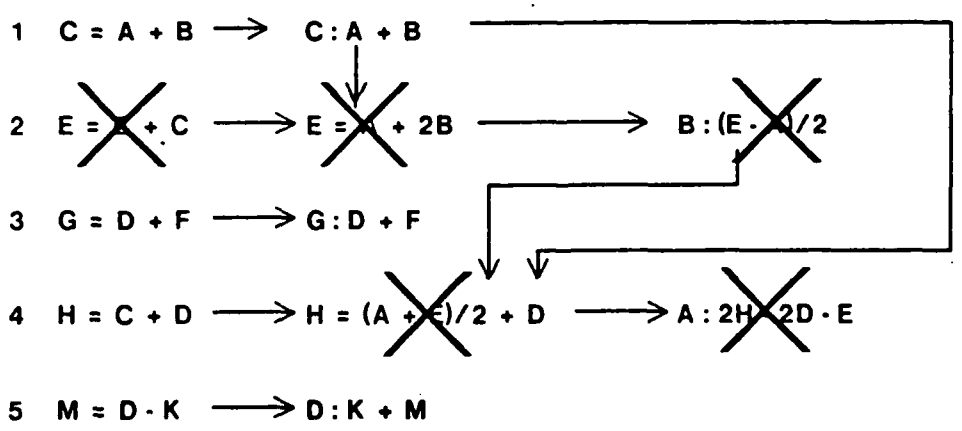


Fig. 17. Algebra Example
Example showing what ORACLE does when equation 2 is retracted.

Equation 4 can be re-solved for a new valid result. The ordering in the following table reflects the fact that equation 5 has already been solved when equation 4 is re-solved.

	Equation	Equation Solved	Variable-Value	Equation Truth	Value Truth
1	C = A + B	C = A + B	C : A + B	True	True
2	E = B + C	E = A + 2B	B : (E - A)/2	False	False
3	G = D + F	G = D + F	G : D + F	True	True
5	M = D * K	M = D * K	D : K + M	True	True
4	H = C + D	H = A + B + K + M	A : H-M-K-B	True	True

Note the substitution for B is not made and B is not considered to have a known value.

4.2.2 Equation Solving

The heart of the ORACLE is a data-base which keeps track of the analysis results. Each entry in the ORACLE data-base is a collection of four properties associated with the solution of a variable's value and indexed by the variable's name. The four properties are the **CURRENT-VALUE**, **TRUTH-VALUE**, **REASONS**, and **CONSEQUENCES**. The **CURRENT-VALUE** is the last value calculated for this variable. The **TRUTH-VALUE** is **IN** if the current value is valid, and **OUT** if the variable's value has been retracted since it was last solved for. **REASONS** is the list of equations used to calculate the variable's value. The first reason in the list is the name of the assertion that directly caused this variable to be solved for. The other reasons represent the dependency of the current value on other variables solved before this variable. **CONSEQUENCES** is a list of assertions that used the current value of the variable when solving their equation.

CIROP maintains its own data-base of assertions which includes the equation assertions. Properties of an equation assertion are the equation and the name of the solved variable. The assertions used in the previous example are as follows:

```
ASSERTION1 : c = a + b
ASSERTION2 : e = b + c
ASSERTION3 : g = d + f
ASSERTION4 : h = c + d
ASSERTION5 : m = d * k
```

After solving these equations the data base is as follows:

Variable-Name	Value	Truth	Reasons	Consequences
C	A + B	IN	(ASSERTION1)	(ASSERTION4 ASSERTION2)
B	E/2 - A/2	IN	(ASSERTION2 ASSERTION1)	(ASSERTION4)
G	D + 1	IN	(ASSERTION3)	None
A	2E-2D -1	IN	(ASSERTION4 ASSERTION2 ASSERTION1)	None
D	K + M	IN	(ASSERTION5)	None

The most interesting entry is the second entry. This entry represents the solution of the equation associated with ASSERTION2 which was solved for the variable B. The value found was (E/2 - A/2). The solution depended on ASSERTION1 and ASSERTION2. The solution of the equation associated with ASSERTION4 depended on this value of B. After equation 2 is retracted and equation 4 is re-solved the data-base looks like this:

Variable-Name	Value	Truth	Reasons	Consequences
C	A + B	IN	(ASSERTION1)	(ASSERTION4 ASSERTION2)
B	E/2 - A/2	OUT	(ASSERTION2 ASSERTION1)	(ASSERTION4))
G	D + 1	IN	(ASSERTION3)	None
A	E - B - K - M	IN	(ASSERTION4 ASSERTION5 ASSERTION1)	None
D	K + M	IN	(ASSERTION5)	(ASSERTION4)

4.3 Procedure used in ORACLE system

The general procedure of the algebra system is as follows:

Given a queue of assertions representing equations and a data-base of known variables, repeat the following steps until the queue is empty.

1. Take the first assertion off the queue and call it A. If the truth value of assertion A is IN, get the EQUATION property of assertion A to be used in the following steps. If the truth value of assertion A is OUT, ignore steps 2 and 3.

2. Until every variable in the equation is unknown, substitute the value of the first variable with a known value. For each substituted variable, remember the name of assertion A on the consequence list of that variable. Also, get the name of the first reason associated with each substituted variable.

3. Solve the resulting equation in terms of one of the unknown variables V. Enter this solution into the ORACLE database as the value of the variable V. Set the truth value of the entry in the ORACLE data-base to IN. Add the name of the assertion A to the front of the list of reasons collected in the last step and enter

that new list as the reason property for variable V in the ORACLE data-base. Remember the name of the variable V as the **Solved-Variable** property on the assertion A.

4.4 CIRCOM

The heart of the algebra system is a simple equation solver, written by G. Sussman as part of his CIRCOM system, that does simple substitution and simplification. To solve equations, it chooses a variable to solve for, and then tries to isolate that variable. Expressions are reduced to a rational form which is a ratio of two relatively prime multivariate polynomials. The mathematical operators used are the basic arithmetic operators: add, subtract, multiply, and divide. The rational form is complete in that any two expressions that are equal have the same rational form.

4.4.1 Retracting Equations

To use the ORACLE algebra system in a non-deterministic system, it must be able to undo the affects of invalidated equations. The system keeps track of the consequences for each variable. Here is the procedure for retracting an equation.

To retract equation E with associated assertion A, do the following.

1. If this is the original assertion to be retracted, set the truth value of the assertion to **OUT**; otherwise, add the assertion to a queue to be re-solved after this procedure is finished.

2. Get the name of the variable V that was solved for when the equation was originally solved. This is the **Solved-Variable** property on the assertion A.

3. Find the entry associated with the variable V in the ORACLE. Set the truth value of this entry to **OUT**.

4. Get the list of consequences in the entry. This is a list of assertions representing other equations which depend on the now obsolete value of variable V.

5. For each assertion A on this list of consequences, execute the entire procedure from the step 1. This process eventually terminates, since all paths terminate at some variable with no consequences. Consequences can only point to equations that were solved after themselves, so there can be no circular paths.

When this procedure is finished, it produces a queue of valid equations that must be re-solved. The normal equation solving procedure is invoked on this queue.

5. Controlling the Design Process

An engineer uses strategic knowledge to control the design process. CIROP mimics the human engineer by using strategic knowledge in two phases of the design process. Strategic knowledge guides the design of the initial prototype. If the prototype fails to meet its specifications, strategic knowledge suggests alternative ways to improve the circuit.

5.1 Guiding the Prototype

The experienced engineer uses learned heuristics to guide the prototype design of a circuit. Usually more than one rule exists that can be used to expand an abstract object. The learned heuristics are used to select an appropriate rule from the possibilities to expand the object. The primary heuristic is to keep the design as simple as possible. In some situations the simplest design may be obviously inadequate. The reasons for preferring one stage over others are usually based on tradeoffs due to the given specifications.

5.1.1 Simplest First

The writer of the rules can include a subjective measure of the rules complexity. This measure is distilled to a number by the writer, where a lower number represents a simpler object. When expanding an object, the applicable rules are sorted based on this measure, simplest ones first. In practice, this measure can be used to state a preference for using one rule over another regardless of their relative complexities.

5.1.2 Tradeoffs

No design exists for an all-purpose operational amplifier. Many different operational amplifiers exist to serve varying requirements. Each operational amplifier is thus designed with finite values for the important specifications appropriate to the need for the amplifier. The differences between operational amplifiers are based on tradeoffs between specifications. For instance, if input bias current is a high priority while the slew-rate is not, that amplifier's design differs from an amplifier requiring a high slew-rate. This tradeoff between the input bias current and the slew-rate of an operational amplifier results in different operational amplifiers, each depending on their requirements.

Tradeoffs can sometimes be expressed as simple equations. These equations are approximate models of

the behavior of the type of object that one is building and can be traced to the more detailed equations that describe the circuit accurately. For instance, a front-end differential pair has an important tradeoff between gain and the input bias current. Detailed analysis shows a direct tradeoff between the gain and input bias current, because the gain depends on the *gm* of the transistors, which depends on the collector current, which depends on the base current, which is the input bias current.

This type of dependence is valid for all normal varieties of differential pairs that use bipolar transistors, except for the current cancellation stage.[1] Thus one can develop a feel for this tradeoff which determines which object in a set of front end stages might be appropriate. These tradeoffs can be used when using a differential pair to decide immediately whether it is worthwhile before expanding it. If the tradeoff is not met, the work of expanding an obviously inadequate circuit has been saved.

One should realize that these constraints are only heuristics, and an applicable constraint may not exist at each level. For instance, the second stage of an operational amplifier will have little direct effect on the input bias current in a well-designed amplifier. Therefore, the second stage would not use a heuristic to constrain itself on the first pass of the design process.

Another problem is that of breaking down constraints. The gain of the entire operational amplifier depends on the product of the gain of the first and second stages. Since one does not know the gain of either stage, it seems hard to pass along a reasonable constraint to the lower stage. A constraint could be passed to the first stage that contained the second stage's contribution to the gain symbolically. Then the first stage would further constrain the second stage gain. When the second stage is created it checks to see if the gain is reasonable. At this point, it is unclear which stage should be replaced. How does CIROP know which stage is at fault if the gain is not met?

Another method notices that the gain will be shared between the first and second stage nearly equally. An approximate constraint can be passed to each stage, which is the square root of the total gain. Then the first stage has real numbers for gain and bias current. Using this, CIROP can check to see if the first stage is reasonable. If the first stage is not reasonable, CIROP goes to the next appropriate rule for building the first

1. The current cancellation stage purposely breaks the direct connection between the gain and the input bias current.

stage.

Any analysis that occurs before the circuit is completely designed may be faulty since it is based on assumptions of what the undesigned parts will do. It is reasonable to assume that the analysis, while possibly faulty, is nearly correct. At this phase of the design it is important that no possible solutions be eliminated before being given a fair chance. Faulty analysis could eliminate good solutions, while allowing bad solutions to be tried. This may occur because certain heuristic constraints model the behavior too exactly. This problem is solved by ensuring that heuristic constraints always use optimistic models of the behavior, thus erring on one side only. The heuristics may allow bad solutions to be tried, without eliminating good solutions. For instance, using the gain example, the first stage gain is checked, assuming the second stage can provide an optimistic amount of gain. Any doubts about the second stage contributing enough gain are well founded, since the first stage is assumed to provide as much gain as possible.

The affect of the specifications on the design is illustrated in Figure 18. As the gain/bias-current tradeoff increases, it is more difficult to satisfy for a given circuit. With a low tradeoff, any circuits that CIROP can synthesize will satisfy the tradeoff. As the tradeoff increases, fewer circuits can satisfy the tradeoff, until no circuits can satisfy the tradeoff. Note: the number of possible circuits does not fall exactly to one before hitting zero. Even when the tradeoff is very difficult to meet, there are several options. For instance, the biasing of the output stage will have little effect on the gain. Therefore, several operational amplifiers can be built using different biasing on the output stage, although they are identical otherwise and barely meet the tradeoff.

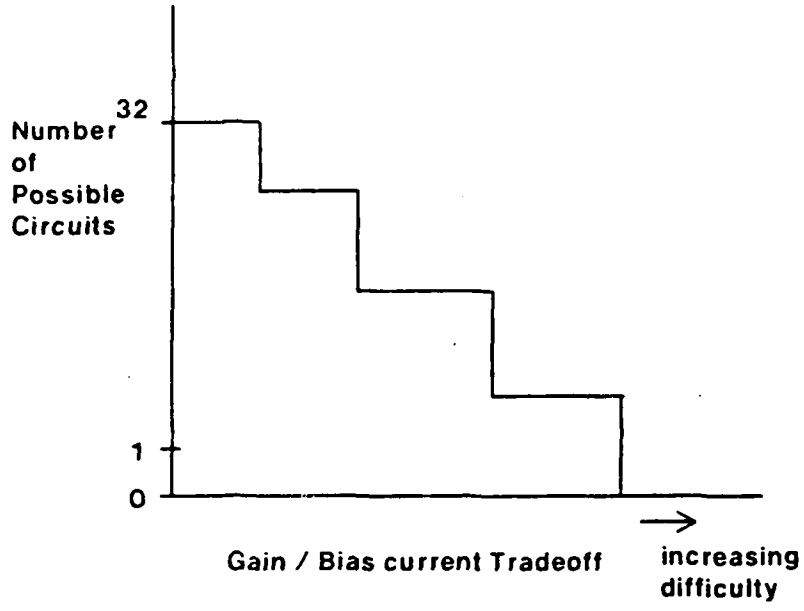


Fig. 18. Tradeoff of Specifications
The number of possible circuits that CIROP might design decrease as a tradeoff becomes more difficult to satisfy.

5.2 Analysis and Testing

CIROP uses the ORACLE to solve the equation assertions made during the creation of the circuit. During analysis, the ORACLE finds values for most of the variables. The top level specifications are compared with the specifications of the proposed circuit. Since some of the specifications are interdependent, they cannot be checked individually. The ORACLE can be queried for the circuit's value of a specification. If the value is a number, it can be compared directly with the goal specification. If the value is an arithmetic expression, it depends on one or more of the other specifications. One by one, the other goal specifications are asserted as necessary until the arithmetic expression becomes a number. The resulting temporary assertions are retracted as soon as a numeric value is found for the original specification. For example, after a simple operational amplifier is created, the gain and bias current have the following values.

```
GAIN =          (* -2.9990628e7
                   LOAD-RESISTANCE
                   (CURRENT (BASE (Q (POS-Q (FIRST-STAGE OP-AMP))))))

BIAS-CURRENT = (* -1. (CURRENT (BASE (Q (POS-Q (FIRST-STAGE OP-AMP))))))
```

To check the gain specification, we must know the values of the variables LOAD-RESISTANCE and (CURRENT (BASE (Q (POS-Q (FIRST-STAGE OP-AMP))))). The variable LOAD-RESISTANCE is a specification and the variable (CURRENT (BASE (Q (POS-Q (FIRST-STAGE OP-AMP))))) depends on the bias current specification. The gain can be found by asserting values for the unknown specifications as follows:

```
assert LOAD-RESISTANCE = 2000.
assert (BIAS-CURRENT OP-AMP) = 2.0e-7

; the new values.
(CURRENT (BASE (Q (POS-Q (FIRST-STAGE OP-AMP))))) = - 2.0e-7
GAIN     = 11996.
```

The circuit gain has been reduced to a numeric value and can be compared against the goal specification for the gain. Each specification is checked in this manner.

5.3 Failure Rules

A designer's response to failure depends on his experience. An experienced designer has learned characteristic ways that circuits fail, and methods for improving circuits which do not work for well defined reasons. An inexperienced designer analyzes in detail, hoping to find the error. Paradoxically, it is easier to mimic the experienced designer than the inexperienced. Detailed error analysis of a circuit requires complex qualitative thinking, which is not formalized well enough to use in a program. Even human designers find it difficult to analyze circuits in detail and often resort to approximate models.

CIROP's failure control rules embody the strategic knowledge used by experienced designers to improve circuits. A failure control rule has a pattern of applicability that specifies which phrase rule failed, and which specifications were not satisfied. The body of the control rule directs the interpreter to try other phrase rules based on the knowledge embedded in the control rule. The following example shows the knowledge that indicates that if the gain and bias current cannot be met with a simple-differential-pair, the

differential pair should be redesigned as a current-cancellation-amplifier.[1]

```
(in-case-of-failure-of
   simple-differential-pair
   ((gain gm)(bias-current gm))
   (try current-cancellation-amplifier))
```

1 The electrical reasons behind this rule are explained in Section 5.5.2

5.4 Backtracking with Failure Rules

Backtracking occurs when a specification is not met. A list of equation assertions which were used to determine the specification's value is contained in the ORACLE. From these equations, CIROP finds the objects that asserted them. CIROP also finds the superiors of each object. These objects represent the possible parts of the circuit that may need redesigning. For each part, CIROP finds the applicable failure control rules associated with the deficient specifications. The **use** property of the part is also compared with the failed specifications. From these a list of applicable failure rules is isolated. This process of finding applicable failure rules is repeated on each part that was found.

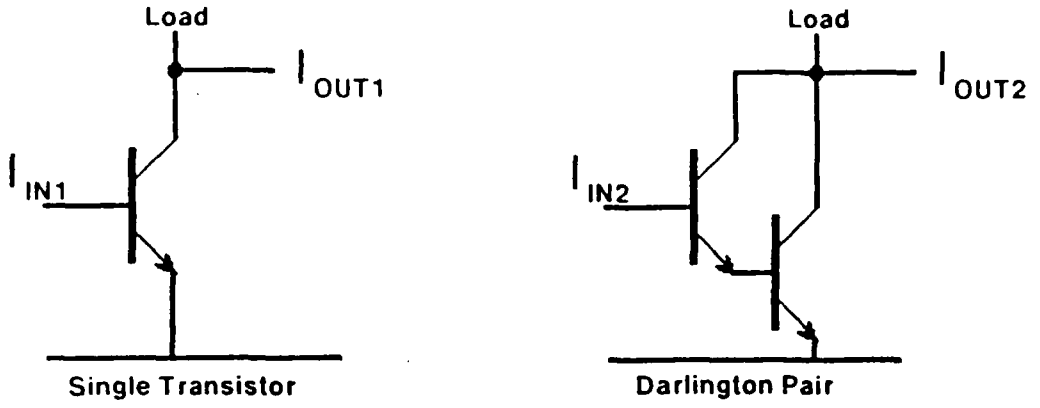
CIROP randomly chooses a failure rule from the applicable rules. The offending part,that the failure control rule matched is revoked, and all the assertions dependent on that part are removed. Any objects that were created by the offending part are also removed from the circuit. The phrase grammar rule suggested by the failure rule is used to rebuild the part. If the part is not a terminal object, its internals are built by the original procedure for creating objects.

5.5 The Knowledge used to Develop Failure Rules

The following subsections describe some of the knowledge that one must distill to create the failure rules.

5.5.1 Improving the Common Emitter's Current Gain

The second stage of an operational amplifier often uses a common emitter amplifier for current gain. Figure 19 shows the difference between implementing a virtual transistor as a single transistor and as a darlington pair. The current gain for the single transistor is only β, while the current gain for the darlington is β^2. Based on simplicity, the single transistor is preferable, however, if it does not supply enough gain, it can be replaced with a darlington pair. Therefore, one can derive the failure rule described in Section 5.3.



$I_{out1} = \beta * I_{in1}$

$I_{out2} = \beta^2 * I_{in2}$

Fig. 19. Current Gain in Common Emitter Transistor
Current Gain in a Common Emitter Amplifier can be increased by increasing the β.

5.5.2 Improving the First Stage Transconductance

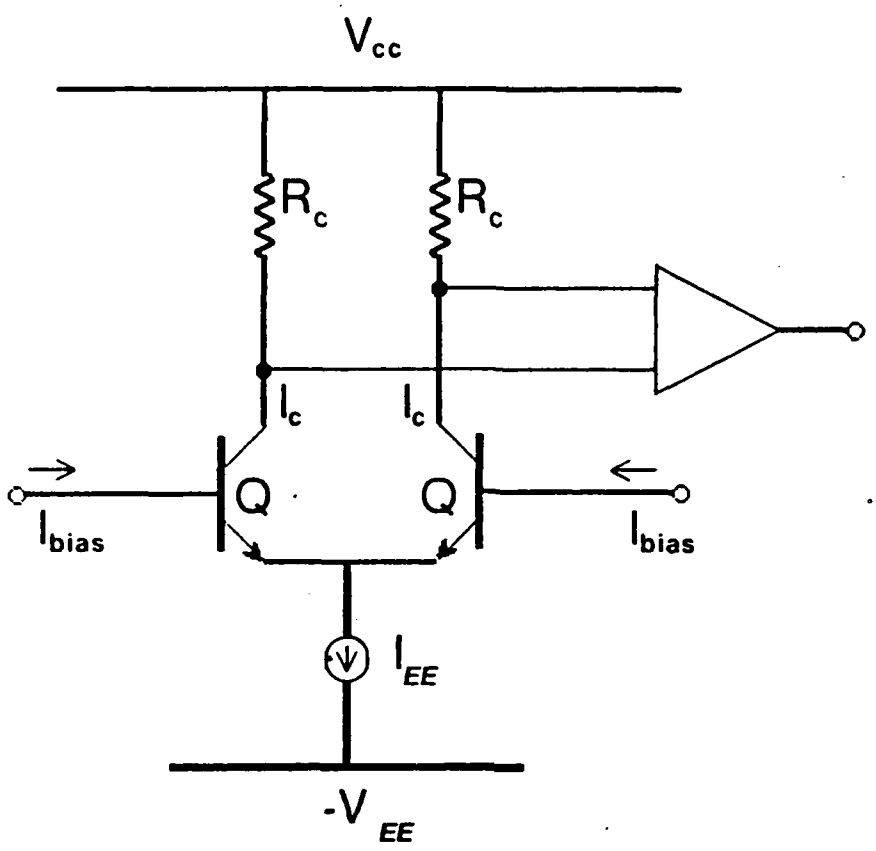
The transconductance of an operational amplifier's first stage[1] is usually dependent on the quiescent collector current as shown in Figure 20.

The transconductance can be increased by increasing the collector current. Unfortunately, this also increases the input bias current. To increase the collector current without increasing the input bias current, the effective β can be increased, or the topology can be changed to break the direct dependence of the collector current on the input bias current.[2]

Following are some ways the input bias current can be lowered.[3]

---

1 This discussion assumes that the operational amplifier's first stage is a bipolar differential pair.
2 This is not meant to be an exhaustive listing of methods to increase the transconductance.
3 Unless specifically stated, the discussion of topologies assumes the first stage is a bipolar transistor emitter coupled pair (a differential pair).



$I_c = \beta * I_{bias}$

$gm = \frac{qI_c}{kT}$

Fig. 20. Input Bias Current Example
Simple Differential Pair for the first stage of an operational amplifier, showing relation between input bias current, the quiescent collector current and the transconductance.

Since the base current is directly proportional to the collector current by the factor $1/\beta$ of the transistor, the input bias current is lowered most easily by lowering the quiescent collector current of the input transistors. This method works as long as the collector current is not too low for other specifications to be met. For instance, a low collector current can lower the frequency response. The overall gain of an operational amplifier has the *gm* of the first stage as a major contribution, but this is directly proportional to the collector current in the first stage.

Another method of lowering the input bias current is to increase the effective β of the first stage input

transistors. This permits a lower base current for an equivalent collector current. The β can be increased by using super-beta transistors. A super-beta transistor has a very narrow base region which improves its base transport factor and its emitter efficiency. This increases the β by as much as a factor of ten. The narrow base region also causes the transistor to have a very low breakdown voltage across this region. All the transistors in the operational amplifier circuit cannot be super-betas, because the circuit would not withstand the required operating voltages. Therefore, the super-beta transistor must be used in a configuration that prevents it from seeing large voltages across its terminals.

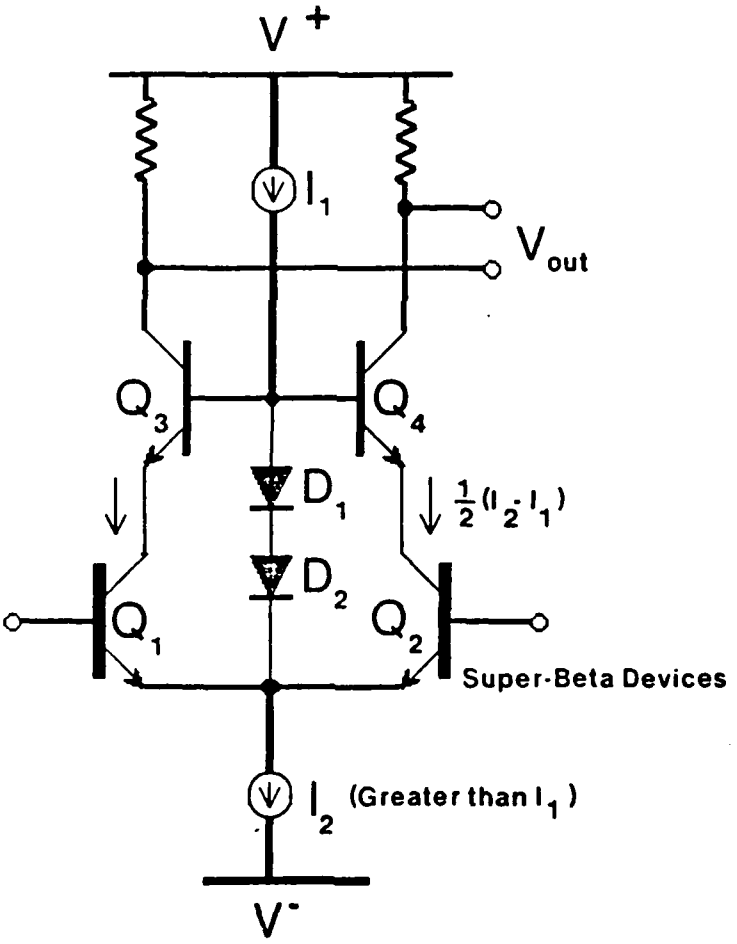


Fig. 21. Super-beta Differential Pair
Super-beta transistors are used in the differential pair to increase the β of the first stage. This can decrease the Input Bias Current while keeping the stage's *gm* constant.

In the detailed example of a super-beta circuit in Figure 21, the elements D1, D2, Q1 and Q3 form a loop. Therefore, KVL forces Q1's collector to emitter voltage to be approximately the voltage drop across one of the diodes. Since this averages six-tenths of a volt, the voltage across the input transistor is kept at a small constant. Super-beta transistor Q2 is protected in the same way. Also, two back-to-back parallel diodes connect the two inputs to protect the transistors from large input voltage differences.[1]

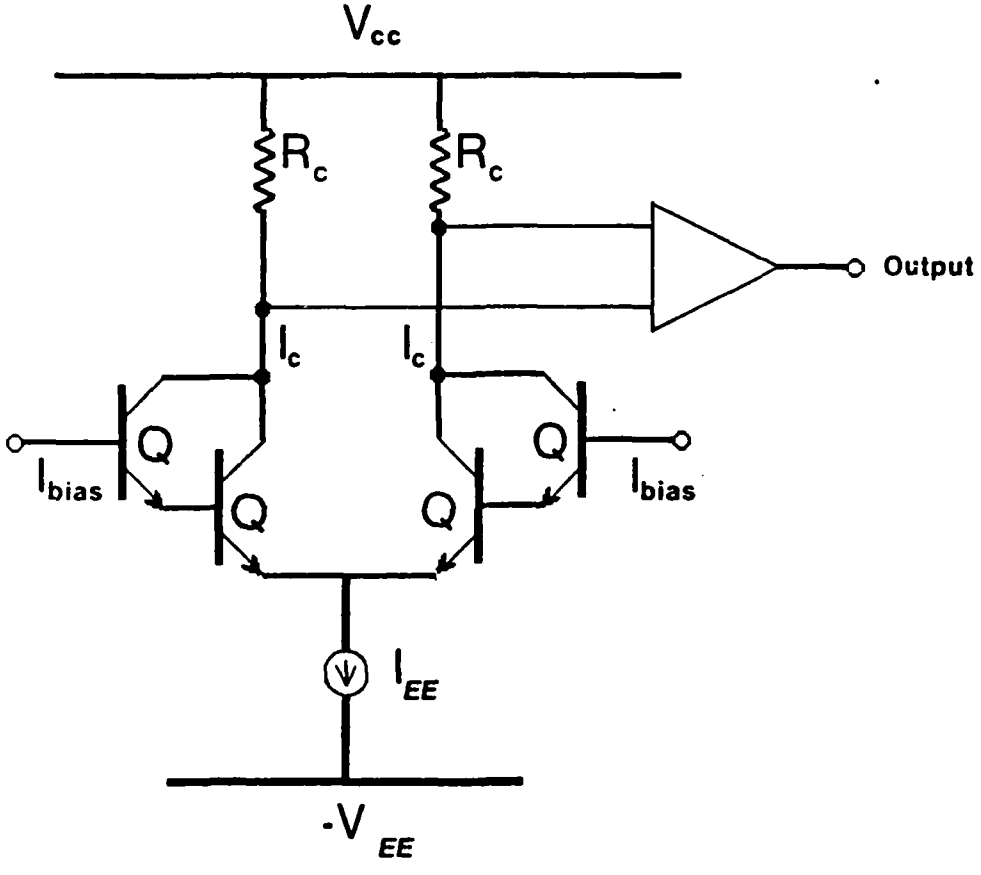


Fig. 22. Darlington Differential Pair
Darlington transistors can be used in the differential pair to increase the β similar to the super-beta transistors.

1 The diodes are not shown to preserve clarity.

Another way of increasing the effective β of the first stage input transistors is to use darlington pairs instead of single transistors, as shown in Figure 22. A darlington pair consists of at least two transistors connected so as to approximate a single transistor with a β approximately the product of the individual β's. However, darlingtons have disadvantages. They have more circuit components in the signal path of the first stage, which can increase the first stage's asymmetry, drift, and noise.

6. Detailed Operational Amplifier Example

This chapter revisits the design example of Chapter 2, detailing CIROP's design decisions.

6.1 Detailed Scenario

The initial goal is to build an operational amplifier with the following NEW-PART assertion and specifications.

```
(new-part op-amp
          ((type op-amp amplifier)
           (has op-amp (input voltage))
           (has op-amp (output voltage))
           (has op-amp (input differential))
           (has op-amp (output single-ended))))
```

gain	500000.
input-offset-current	10 nanoamps
slew-rate	0.2 volts per micro-second
output-drive-current	15 milliamps
unity-gain-frequency	3 Megahertz
output-load-resistance	2 kohms
input-bias-current	0.2 microamps

The first action is to find the rules that match this NEW-PART assertion. For such a high level object, different rules would probably describe major differences in the strategy if designing the circuit. For instance, the rule used might describe the most common operational amplifier strategy with three stages, while others might describe four stage operational amplifiers. The standard rule for creating an operational amplifier is as described in section 3.7.3. The following concepts are implicit in the rule definition: The first stage will convert the differential input to single-ended. The middle stage will provide any necessary gain that the first stage cannot supply. The third stage will buffer the output. The feedback stage will be used to make the operational amplifier stable at frequencies of interest.

The result of the top level rule is to add four NEW-PART assertions to the queue, connect these new parts to the abstract operational amplifier, and create several constraints between the objects. The original NEW-PART assertion that created the top-level object has been removed from the queue.

The new first assertion on the queue is the assertion for the first stage of the operational amplifier as

follows:

```
(new-part first-stage
          ((type first-stage amplifier)
           (has (input differential))(has (output single-ended))
           (has (input voltage))(has (output current))
           (has (sign pnp)))
          ((use . ((gain gm) (slew-rate gm)))
           (with-specs
            (= (transconductance (first-stage))
               (sqrt (* 2 (gain)))))))
```

This matches the pattern of the following rules: simple-differential-pair, current-cancellation-pair, and super-beta-differential-pair.[1] Since the simplest of these rules is the simple-differential-pair, it is checked first. The simple-differential pair has a *with-specs* constraint that relates the transconductance to the gain. This constraint is temporarily asserted and the specifications are checked to see if they could be met with this constraint. Since the constraint is satisfied the proposed simple-differential-pair first stage is adequate for now.

The assertions in the simple-differential-pair's body are asserted, which adds more parts to the end of the queue and more constraints are created. The assertion used to create the first-stage is removed from the queue.

The new first assertion on the queue is the top-level rule's NEW-PART assertion for the second stage as follows:

```
(new-part second-stage
          ((type amplifier)
           (has (input single-ended))(has (output single-ended))
           (has (input current))(has (output voltage))
           (has (sign npn)))
          ((with-specs
            (= (transresistance (second-stage ?op-amp))
               (sqrt (* 2 (gain ?op-amp)))))))
```

The only rule that matches this is a common-emitter amplifier. The common-emitter amplifier is then checked to see if it meets the specifications. The rest of the second stage and the third stage are designed in similar ways. The next few paragraphs concentrate on the design of the first stage down to the level of physical transistors.

After building some of the rest of the circuit, it will eventually get to the NEW-PART assertions that

1. The rules for parts are all in Appendix 1.

were created by the simple-differential-pair rule. The first such assertion is the following:

```
(new-part matched
          ((type matched-pair)
           (has (sign ?sign))))
```

A matched pair is a pair of transistors that have been carefully manufactured so that their physical properties are nearly identical. Using a matched pair for the two transistors in the emitter coupled portion of a differential pair improves the symmetry of the stage. The implementation of the matched pair transistors is a pair of pnp bipolar junction transistors.

The rest of the circuit is designed in the same manner. The complete prototype is analyzed and compared to the specifications. At this point in the design, the gain and input bias current are the following:

```
GAIN =                  (* -2.9990628e7
                           LOAD-RESISTANCE
                           (CURRENT (BASE (Q (POS-Q (FIRST-STAGE OP-AMP))))))

INPUT-BIAS-CURRENT =(* -1. (CURRENT (BASE (Q (POS-Q (FIRST-STAGE OP-AMP))))))
```

Given the specified input-bias-current and load-resistance, the maximum gain will be 12000. This does not meet the specification. CIROP must backtrack by redesigning some part of the circuit.

The calculation of the circuit's gain is dependent on forty-three of the total constraints asserted. The parts that created these constraints are found. Only three of these parts have a gain *use* property and also have appropriate failure rules for improving the gain. These three parts are the first stage, the first stage matched pair, and the second stage transistor. To improve the gain, one of these options must be rebuilt. The first choice is to replace the second stage's single transistor with a darlington pair. Associated with the second stage's transistor is the following failure rule:

```
(in-case-of-failure-of
   single-transistor-bjt
   ((gain beta))
   (try double-darlington-transistor-bjt))
```

The virtual transistor used in the second stage is removed from the circuit. This also removes the physical npn transistor that was the only part of the virtual transistor. Removing the parts invalidates any constraint equations that they had asserted. As a result, only a few other equations must be re-solved, those that used the values derived from the now invalidated equations. After this, the failure-rule is invoked to

determine how the virtual transistor will be rebuilt. In this case, it will create a darlington pair instead of a single transistor. The improved circuit is analyzed and now meets the specifications. To re-analyze the circuit, only the new constraints asserted by the new parts must be solved.

7. Related Work

7.1 EL

"EL " is a rule-based system for computer-aided circuit analysis [Stallman]. Given a circuit description, EL determines the state of the active elements and the values of the voltage potentials and currents at all nodes and branches of the circuit. EL uses *propagation of constraints* and *dependency-directed backtracking* to control the analysis. Since it is an analysis program, it does not need to understand the creation of topologies. It is given a fixed topology.

EL represents circuit specific knowledge as assertions in a relational data base. The general knowledge about circuits is represented by "LAWS", which are demons subject to pattern directed invocation. A new assertion into the data base triggers matching demons. Triggered laws are put onto a queue along with information that tells what part of the circuit they will operate on. The demons are executed when taken off the queue. Executing a demon has one of two useful actions: it either makes a new assertion into the data base, which may restart the whole process by matching more demons, or it may discover a contradiction.

EL makes assumptions about the states of the active devices. Since these assumptions are originally guesses, they are frequently wrong. If an assumption is wrong a contradiction will arise, which is a set of assertions which can not all be true at the same time. When a contradiction is discovered, EL invokes a method called *dependency-directed backtracking*. This automatic procedure removes one of the assumptions associated with the contradiction, and tries another assumption. This leads to more effective control of combinatorial search than undoing the last assumption which was made.

EL needs backtracking since it makes guesses about the transistor's states. CIROP needs backtracking since it makes guesses about how to refine abstract circuit objects. A combination of the *use* property and the failure control rules are used to determine which assumption must be undone when CIROP finds a contradiction (a failure to meet specifications). The failure rule also tells which option should replace the faulty guess.

CIROP's ability to solve algebraic equations provides the same function as *propagation of constraints*. *Propagation of constraints* can be viewed as a system that solves equations by always solving the equations

with the fewest unknowns. This imposes an order on the equations to be solved. CIROP is more general than that, since it does not require the equations to be solved in a particular order.

7.2 SYN

SYN [deKleer-3] shows that *propagation of constraints* is useful in synthesis, as well as analysis of electrical circuits. EL assumes the component values are known and calculates the voltages and currents of interest. SYN shows that the calculations can go in the other direction: given enough constraints about voltages and currents, one can solve for the component values. By stating goals such as gain or particular currents and voltages, SYN synthesizes the circuit by finding the values for the components. For example, in Figure 23, EL can solve for the current I_1, given the voltage and the resistor values. SYN does that, plus solves for resistor values, given the desired currents and voltages.

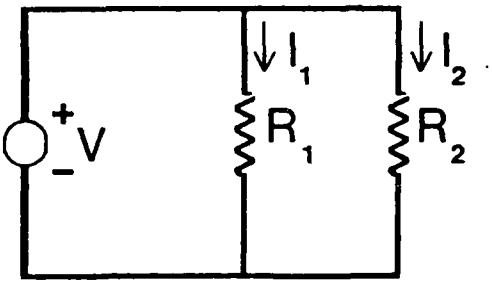


Fig. 23. Example for EL and SYN
EL can solve for the currents labeled I_1 and I_2 given the values of R_1, R_2, and V. SYN can also solve for R_1 and R_2 given values for I_1, I_2, and V.

To manage the complexities of synthesizing a circuit, an engineer simplifies the problem by constructing separate models of the circuit. Each model describes some aspect of the circuit's behavior. Each model is a new circuit that can be more easily analyzed then the original, although any one by itself does not model the complete behavior of the circuit. SYN creates separate models similar to that used by a human engineer. For instance, each transistor can be modeled with a bias model or an incremental model as shown in Figure 24.

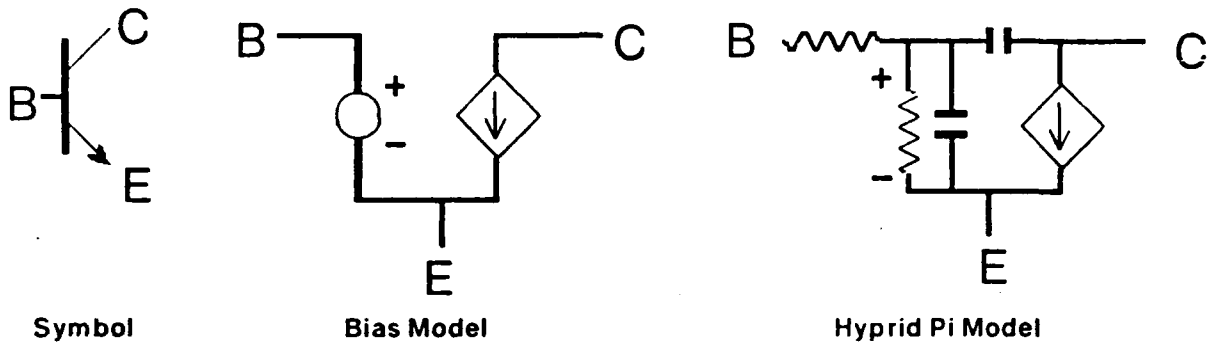


Fig. 24. Transistor models used in SYN
SYN simplifies analysis by using different models of devices for different regions of operation.

Although having different models simplifies the algebra required to synthesize a given circuit, SYN is still limited by algebra when the circuit is large. The circuit in Figure 25 is the largest circuit SYN successfully synthesized.

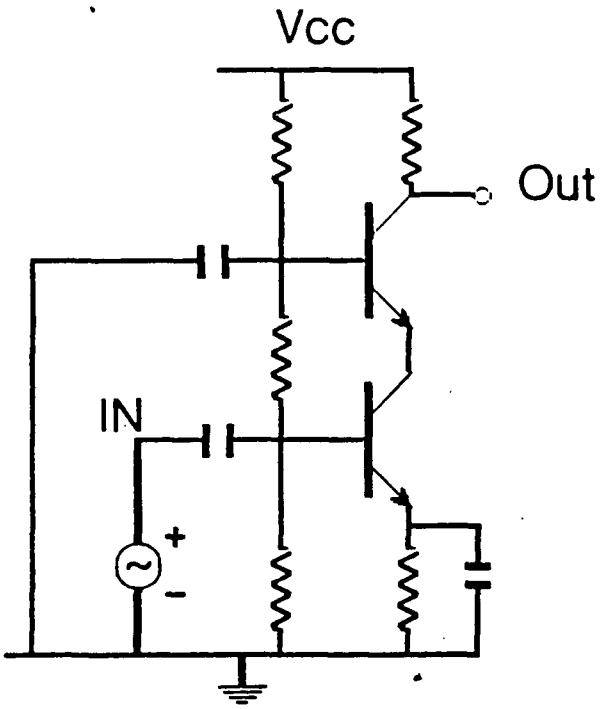


Fig. 25. Cascode Circuit Synthesized by SYN
One of the largest circuits succesfully synthesized by SYN.

7.3 A Simple Model of Circuit Design

Roylance's Master's thesis [Roylance80] is one of the first attempts to creatively design circuits. His system models fundamental rules about capacitors, resistors, operational amplifiers, feedback and network laws. These rules describe the device's causal behavior. The causal behavior of devices suggests strategies for using them in design. Starting with a goal, the rules are used to decide what components to use and how to connect them. The system has no memorized circuit fragments (combinations of more than one primitive device), because it builds them dynamically.

Since the system builds the circuit from basic components, it is not limited in the topologies it uses. It has the potential of creating and designing new topologies to solve a problem. Unfortunately, it cannot take advantage of topologies that are already known to solve specific problems. This approach may be limited in the problems it can solve until a large portion of general circuit knowledge is incorporated as rules. For

example, this approach might have trouble designing a complex current mirror. Although the current mirror in Figure 26 is simple to understand, it is difficult to imagine the knowledge needed to synthesize it the first time. CIROP can use the mirror since it need not know the origins of the circuit fragment to use it effectively and need not synthesize it from primitive elements.

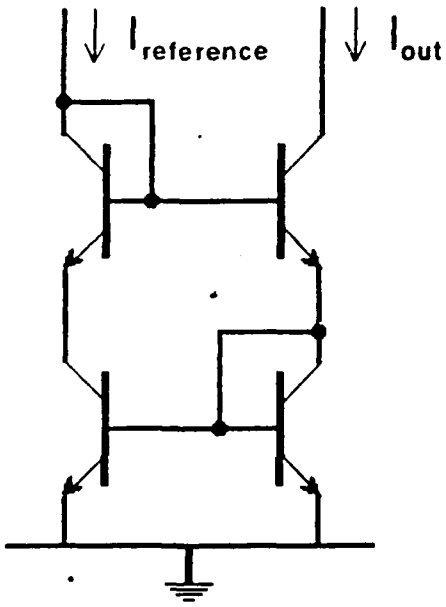


Fig. 26. Current Mirror
Current Mirror that is hard to design from basic principles.

A good circuit design system will need the concepts developed in Roylance's work as well as the concepts used in CIROP. Using only CIROP, one could never design completely new circuits, but using only Roylance's system one must constantly re-invent the wheel. Often a new circuit may be a new twist to an old circuit. CIROP could design the "old circuit" as a first approximation. A system similar to Roylance's, that could understand how the circuit worked and why it failed, could provide the necessary "twist" to improve the circuit to meet the specifications. In this situation CIROP acts as a smart library of circuit fragments with

the knowledge to compose complete circuit prototypes from the fragments.

7.4 Qualitative Analysis

The exact analysis of circuits is very difficult. When a circuit does not meet the specifications it must be analyzed to find the limitations. A simple answer of "No" to the question of "Is the specification met?" is not always sufficient. For instance, if an interaction between sub-parts of an object caused unexpected behavior it would be useful to understand what was wrong. Qualitative analysis of circuits [Williams][de Kleer] can describe in high-level terms how a circuit operates. CIROP would not be able to take advantage of such qualitative analysis. However, a system that combined the ideas of Roylance and CIROP as described in the previous section would understand how the circuit was supposed to operate. The expected behavior could be compared with the behavior predicted by qualitative analysis. This could result in a system that understands the knowledge that is the basis for the failure rules used in CIROP.

7.5 A Refinement Paradigm

The refinement paradigm [Barstow] is a technique that implements a high-level program specification in a low-level language. Several similarities exist between the paradigm and CIROP. The paradigm starts with a high-level program specification. With coding and analysis rules, the specification is refined to a detailed low-level program.

The coding rules are similar to CIROP's phrase grammar rules in that they expand a programming concept into a more concrete procedure. For any particular concept there generally exists more than one rule applicable to expand it. Similar to the circuit grammar, the coding rules describe a refinement tree, which is a space of possible programs.

The analysis rules know about efficiency of implementations, and find upper and lower bounds on alternate implementations of a given concept. These results are used to choose the most efficient coding rule for the given goals. The analysis rules are similar to the *with-specs* constraints used in CIROP's grammar rule patterns. Both affect the decision of which rule is applicable to expand the current abstract object.

The refinement paradigm can be stated as follows:

1. Pick some node of the refinement tree to expand, based on cost estimates for the active nodes (which are non-terminal leaves).

2. Pick some part of the program described by that node to refine, based on the relative importance of different parts.

3. Find the coding rules that can be used to refine that part.

4. Prune those rules that fail to satisfy plausibility requirements.

5. Expand the tree by applying each of the remaining coding rules to create new program description nodes.

6. Compute cost estimates for the new nodes by applying the analysis rules.

The refinement paradigm uses rules which produce "cost estimates" to decide which piece of the program should be refined next. CIROP does not use this type of strategic knowledge for control, although human engineers do. Some circuit designers like to work on the output stage and then work their way back to the input stage. Other circuit designers choose to work on the input stage first. The philosophy is that one should concentrate on the piece of the circuit that is the "most difficult" to build to meet the specifications. The advantage to using such knowledge is that it may direct the designer to a solution more directly. In other words, the designer may have to backtrack from bad solutions less often. CIROP does not use this type of knowledge for two reasons. First, it is very difficult to formalize in the circuit domain. Second, with the failure control rules the amount of backtracking is small. CIROP expands its hierarchy in a left-right breadth-first manner.

Barstow's use of the refinement paradigm does not use backtracking. Since each step of the refinement uses correctness preserving transformations, each program in the refinement sequence will satisfy the functionality specifications. The efficiency rules guide the refinement so that each program in the refinement sequence is more efficient than the next. The only question would be "Is it the best program?" The question's answer can seldom be answered in a definite way, but if the answer is a fuzzy "yes", that is probably good enough. In circuit design the answer is either "yes" or "no" because a circuit either meets the specifications or it does not. Since one cannot verify the specifications exactly until a circuit is built in detail it is possible for the question to be answered "no" even though the next level abstract circuit seems to meet the specifications. If the answer is "no" then the system must be able to backtrack to find a good solution. The

refinement paradigm assumes that using the efficiency rules will at least guarantee a fuzzy "yes" or better. Thus, it is not as important that it be able to backtrack.

7.6 Molgen

Molgen [Stefik] is a program that implements a theory of how to plan gene cloning experiments by refining abstract plans. The important issues in Molgen that relate to CIROP are its use of *Constraint Posting* and *Meta-planning*. Molgen views *Constraint Posting* as three operations. First, constraint formulation is the process of creating constraints. This occurs when abstract objects are refined to more concrete objects. The more concrete object may have constraints that must be met. For instance, when performing a TRANSFORM operation, it is necessary that the bacterium and vector, which are inputs to the TRANSFORM operator, be biologically compatible. Therefore, refining an abstract MERGE operation to a TRANSFORM operation places constraints on the values of the bacterium and vector so the TRANSFORM will work properly. The equations in phrase grammar rules in CIROP are constraints that are formulated when the rule is expanded. The expansion of a CIROP phrase grammar rule is similar to refining an object in Molgen.

Second, constraint passing propagates the constraint across operations to other objects. Subproblems communicate by passing constraints, which is necessary since the subproblems may interact. The constraints on objects are passed through operators. CIROP's objects correspond to the operators in Molgen. The constraints in CIROP are passed when the equations are solved.

Third, constraint satisfaction is the process of looking for concrete objects that will satisfy the constraints placed on an abstract object. If only one object can be found that satisfies the constraints, then Molgen can refine the abstract object to that particular concrete object. If no objects satisfy the constraint, then the abstract object has been overconstrained. CIROP also uses constraints to determine how an abstract circuit object will be expanded. The fundamental difference is that Molgen is a *least commitment* planner. If possible, it will not refine an object until the constraints are satisfied by only one concrete object. CIROP does not try to constrain an object until only one rule is applicable. In circuits, there usually are many ways of satisfying a particular goal so it is not possible to create constraints that will eliminate all but one possibility.

Meta-planning controls the evolution of the plan. Three levels of control are used in to model

hierarchical planning in Molgen. The lowest level are Lab steps, which are the basic operations in the domain. The middle level are Design steps, which select and execute the Lab steps. The top level are strategy steps which select and execute the Design steps.

The Laboratory space represents knowledge about objects and operations in a genetic laboratory. This level is not a control level; it only represents knowledge about genetics. The Laboratory space also contains abstract objects and operations.

The Design space contains knowledge about designing plans. This defines a set of operators for sketching plans abstractly and propagating constraints. Steps are executed in design space to create and refine the laboratory plan.

The Strategy space contains simple knowledge about strategy for guiding the Design space. It represents two major problem solving philosophies: heuristic and least-commitment. The intent is for Molgen to operate in the least-commitment mode whenever possible. If no steps may be taken without making a choice on an underconstrained operation, Molgen will operate as a heuristic planner.

8. Conclusions

8.1 Limitations of a Strict Hierarchy

A hierarchy of circuits can describe any circuit with a known topology. However, it does not always capture the knowledge in a natural way. For instance, the general goal of a current cancellation stage for an operational amplifier's first stage is to dramatically reduce the input bias current. It does this with the clever trick shown in Figure 27. The two current sources labeled I_{bias} supply the current needed to bias the input transistors, which theoretically reduces the input bias current to zero.

The implementation of this trick samples the current necessary to drive the input transistors and

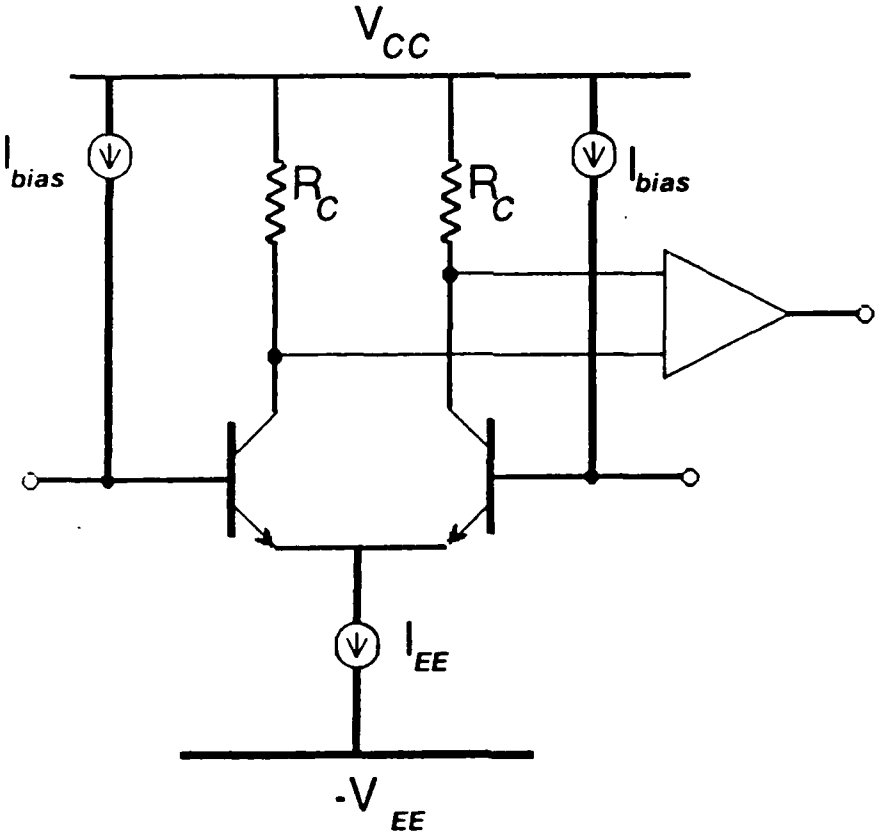


Fig. 27. Current Cancellation Goal
The current cancellation differential pair breaks the direct link between the Input Bias Current and the stage's g_m. This allows the designer to decrease the Input Bias Current substantially.

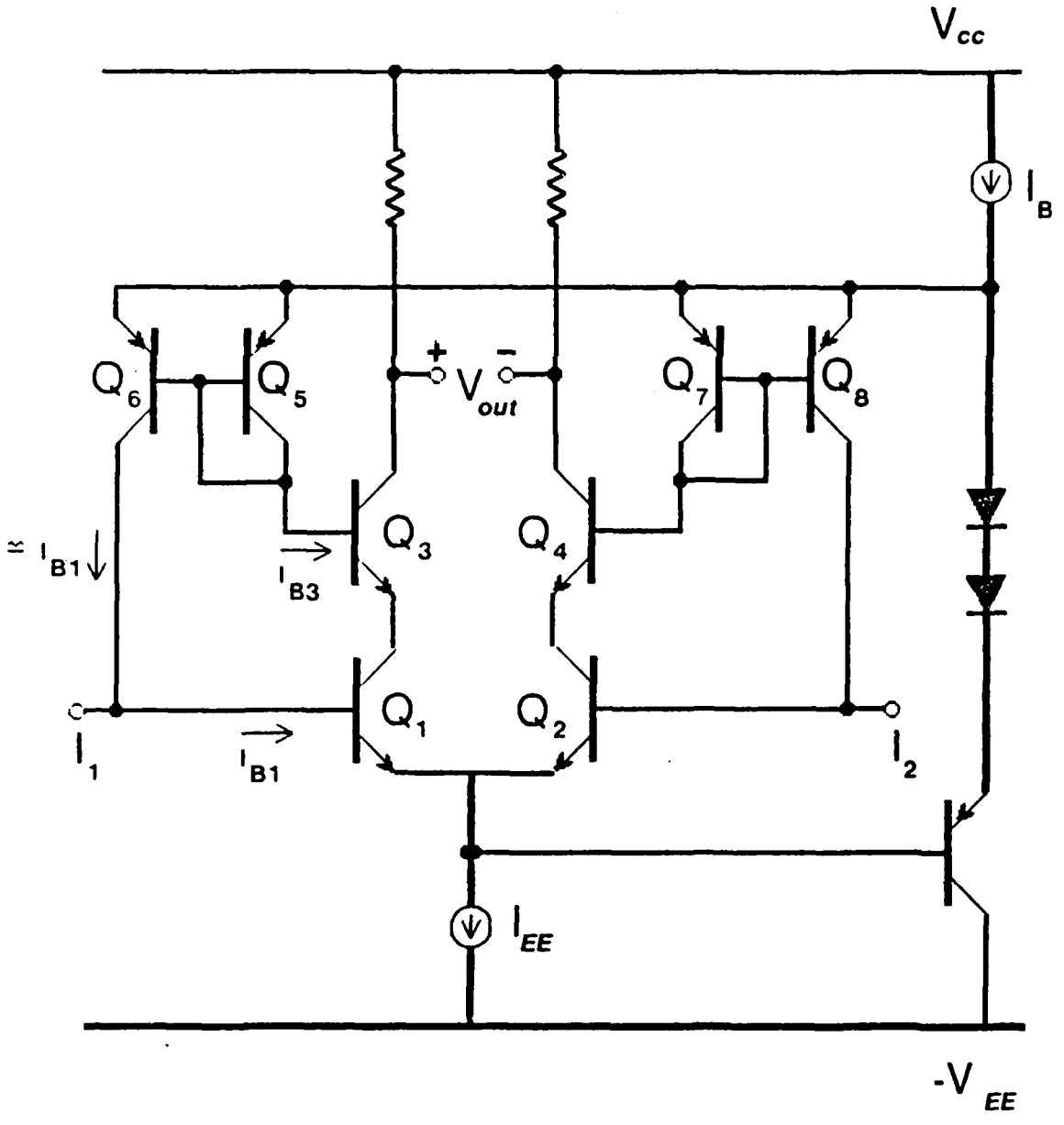


Fig. 28. Details of Current Cancellation
The details of the current cancellation show how the current mirrors sample the collector currents of the differential pairs, and use that to create currents approximately equal to the input transistors' base currents.

supplies it automatically via the current sources. Because of this, the inputs need not supply any current to drive the input transistors. In Figure 28, Q1 is an input transistor, and Q3's collector current is assumed to be approximately the same as Q1's collector current. Assuming the β's of Q1 and Q3 are the same, the base currents of Q1 and Q3 should also be the same. Transistors Q5 and Q6 are configured as a current mirror. They take a reference current, which is the collector current of Q5, and create a matching current as the collector current of Q6. This matched current approximately equals Q3's base current, which approximately equals Q1's base current. This matched current supplies the base current necessary for Q1. The above technique assumes that the transistors' β's are large and nearly equal, which is a reasonable approximation for an integrated bipolar circuit.

Unfortunately, this clever trick does not appear in the circuit grammar representation of the current cancellation stage. Therefore, the entire circuit built around each different use of this trick must be represented in detail in the grammar.

8.2 Transformations

Not all useful circuits can be described easily by a hierarchy of construction rules such as CIROP's phrase grammar rules. For instance, the protection circuitry used in the output stage of an operational amplifier is not really part of the circuit during normal operation. Figure 29 shows a typical output stage before and after protection is added to the top transistor.

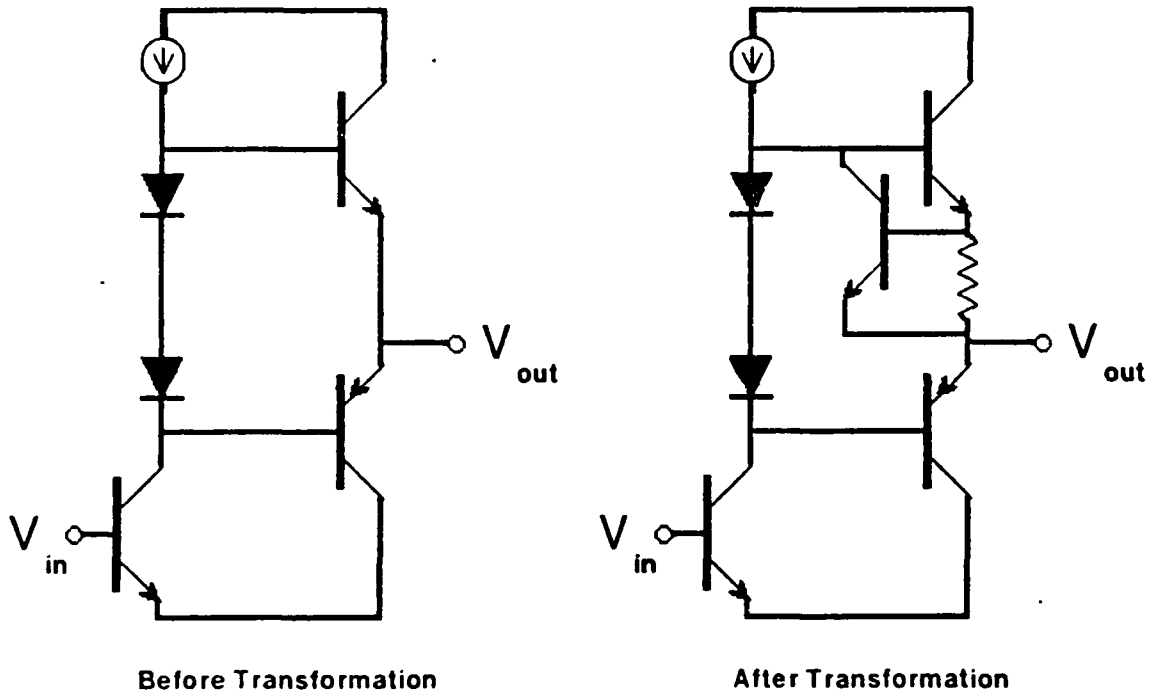


Fig. 29. Output Stage Protection by Transformation
The output stage of an operational amplifier is protected by transforming a fragment of the circuit into a new fragment that still achieves the original goals, but now is protected.

Transformations are context-sensitive rules that can transform an existing part of the circuit into something else. They can help simplify a circuit after it is designed, as shown in Figure 30. The top circuit is the product of the phrase grammar manipulations. The bottom circuit is optimized because the two inductors have been combined into one.

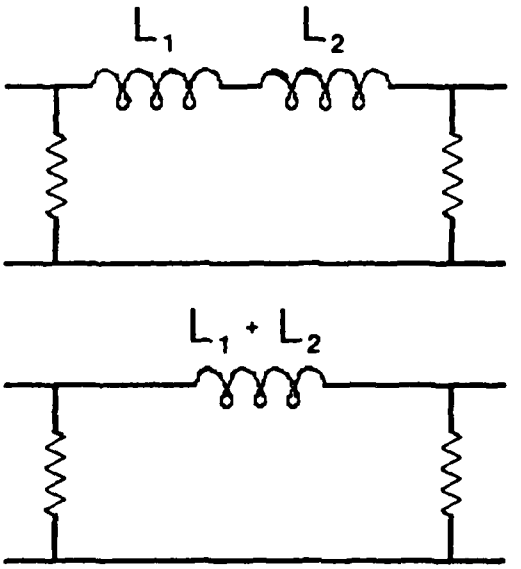


Fig. 30. Inductor Transformation
The example circuits shows a simple optimization of combining two parts into one.

8.3 Algebra

Although circuit designers must seldom solve complex equations, it may still be necessary to solve more complex algebra than CIROP can. Its algebra system is limited because it does not understand inequalities nor approximations, and cannot manipulate any mathematical operations, except the four basic arithmetic operations. Inequalities are necessary since specifications are usually stated as inequalities. Since CIROP treats inequalities as equalities it may conclude that it cannot find a solution for a set of specifications, because it may be harder to meet the specifications exactly. Human engineers use many approximations in design. For instance, the following equation may seem complex:

$$\frac{C * ((\beta + 1) * A + D)}{\beta * A + D}$$

In designing electronic circuits, the variable β often is at least 50-100. It is usually valid to assume that $(\beta + 1)$ approximately equals β. Therefore, the above equation can be simplified to the value C. CIROP

cannot make such simplifying approximations. Without such approximations, a more powerful algebra system is required.

Transistors and other components cannot always be described and analyzed as linear components. This results in equations with exponentials, requiring complex algebraic manipulations. The human engineer can usually simplify equations using knowledge that CIROP does not have.

The algebra system of the future should be able to manipulate more complicated equations, make approximations when useful, and understand inequalities.

8.4 Directing the Search in Circuit Space

The engineer uses his experience to guide him through a design. His experience usually prevents him from going down incorrect paths, and instead leads him to a good solution in a seemingly direct manner. CIROP has approximated this ability to avoid obviously errant paths with the specification tradeoffs mechanism, however, this does not capture all of the intuitive knowledge that humans use. Humans also categorize operational amplifiers into several classes. High-speed, low-power, high-power, and high accuracy are some categories appropriate for operational amplifiers. The differences in the classes are represented by the tradeoffs in the specifications. Knowing the class of a proposed amplifier can direct the search by eliminating options that are never found in amplifiers of that class. CIROP does not use this method, although if the classification knowledge were formalized, it could be incorporated with the strategic knowledge that determines the rules used in expansion.

8.5 Frequency Analysis

Analysis and reasoning about frequency response is so difficult that few human engineers do it well. Reasoning about frequency response requires the ability to analyze the circuit from other viewpoints. Except for slew rate analysis, this area has been avoided in CIROP.

8.0 Summary

The goal of CIROP has been to demonstrate a theory of ordinary design. CIROP does suprisingly well at the task of circuit design. It designs circuits at the level of complexity described in Solomon's classic paper [1974]. Thus, CIROP demonstrates an upper bound on the amount of knowledge needed to achieve this level of competance.

A new grammar describing several varieties of operational amplifiers was developed for CIROP. The phrase grammar's rules were developed to describe the DC characteristics of operational amplifiers. These rules show that each of several common operational amplifiers can be modeled as a hierarchy of abstract objects. The analysis of these circuits was accomplished with the hierarchical assertions of equations.

The representation used in CIROP is general and extensible. New rules can be added that describe new ways of building existing abstract objects and defining new abstract objects. The hierarchical paradigm fits many examples in the circuit domain. For instance, a radio's tuner can be shown hierarchically as a collection of three lower level objects: a converter, an IF strip, and a detector.

There is no direct limitation on the type of equations. If more complex algebra is necessary, one could interface it with a MACSYMA [MACSYMA]. The failure rules could have a more complex interpreter implemented naturally with the rest of the system.

The topology and behavior of the devices are the main ingredients in a circuit. The representation of these should be explicit, which is accomplished by the phrase grammar rules. The equations that model the behavior of the physical objects are the same equations used in models by engineers. The equations used in the abstract objects are also similar to the equations used by engineers and fit the domain naturally. Subject to the limitations discussed, the representation is complete and concise. The rules contain no extraneous information that should confuse an engineer. CIROP accomplished its goal of demonstrating a theory for design.

9. Bibliography

Aho Aho, Alfred V. and Ullman, Jeffrey D. **Principles of Compiler Design** Addison & Wesley, 1979.

Barstow Barstow, David R. and Kant, Elaine. "The Refinement Paradigm: The Interaction of Coding and Efficiency Knowledge in Program Synthesis".

Brokaw Brokaw, A. Paul. **Linear Synthesis for Monolithic Circuits** Analog Devices, Inc., 1978.

Gray P. E. Gray, Paul E., and Searle, Campbell L. **Electronic Principles: Physics, Models and Circuits** John Wiley & Sons, New York, 1969.

Gray P. R. Gray, Paul R., Meyer, Robert G., **Analysis and Design of Analog Integrated Circuits** John Wiley & Sons 1977.

de Kleer1 de Kleer, Johan, J. Doyle, C. Rich, G.Steele, and G. Sussman, "AMORD: A Deductive Procedure System" MIT Artificial Intelligence Lab., AIM 485, September 1978.

de Kleer2 de Kleer, Johan, **Causal and Teleological Reasoning in Circuit Recognition** MIT Artificial Intelligence Lab., AI-TR-529, September 1979.

de Kleer3 de Kleer, Johan, and Sussman, Gerald Jay, "Propagation of Constraints Applied to Circuit Synthesis" MIT Artificial Intelligence Lab., AIM 485, September 1978.

Doyle Doyle, Jon **Truth Maintenance System for Problem Solving** MIT Artificial Intelligence Lab., AI-TR-419, January 1978

MACSYMA The Mathlab Group **MACSYMA Reference Manual** MIT Laboratory for Computer Science., January 1983.

McDermott McDermott, Drew Vincent, **Flexibility and Efficiency in a Computer Program for Designing Circuits** MIT Artificial Intelligence Lab., AI-TR-402, June 1977.

McDonald McDonald, David D., **Natural Language Generation as a Computational Problem: an Introduction** COINS Technical Report 81-33 University of Massachusetts at Amherst.

Modesto Maidique, Modesto A., Sullivan, Douglas R., **Characterization and Analysis of a New High Performance Operational Amplifier Family** Transitron Electronic Corporation, Wakefield, Massachusetts.

Roberge Roberge, James K., **OPERATIONAL AMPLIFIERS: Theory and Practice** John Wiley & Sons Inc. 1975.

Roylance75 Roylance, Gerald L., "Anthropomorphic Circuit Analysis" MIT Bachelor Thesis in EECS, June 1975.

Roylance80 Roylance, Gerald L., "A Simple Model of Circuit Design" MIT Masters Thesis in EECS, 1980.

Shrobe Shrobe, H., **Dependency Directed Reasoning for Complex Program Understanding** MIT Artificial Intelligence Lab., AI-TR 503, April 1979.

Simon Simon, Herbert A., **The Sciences of the Artificial** MIT Press 1969.

Solomon Solomon, James E., "The Monolithic Op Amp: A Tutorial Study" IEEE Journal Solid-State Circuits, vol. SC-9, pp. 314-332, December 1974.

Stallman Stallman, Richard M., Sussman, Gerald Jay, "Forward Reasoning and Dependency-Directed Backtracking In a System for Computer-Aided Circuit Analysis" MIT Artificial Intelligence Lab., AIM-380, Sept. 1976.

Stefik Stefik, Mark Jeffrey, "Planning with Constraints" Stanford Computer Science Dept., STAN-CS-80-784, Jan. 1980

Sullivan Sullivan, Douglas R., Maidique, Modesto A., "Matched Design - Key to Linear ICs" EDN February 15, 1971.

Sullivan Sullivan, Douglas R. and Maidique, Modesto A., **Characterization and Application of a New High Input Impedance Monolithic Amplifier** Transitron Electronic Corporation, Wakefield, Massachusetts.

Sussman-1 Sussman, Gerald Jay, **A Computer Model of Skill Acquisition** Elsevier Computer Science Library, 1975.

Sussman-2 Sussman, Gerald Jay and Steele, Guy Lewis Jr., "Constraints: A Language for Expressing Almost-Hierarchical Descriptions" MIT Artificial Intelligence Lab., AIM-502A, August 1981.

Sussman-3 Sussman, Gerald Jay, "Slices: At the Boundary Between Analysis and Synthesis" MIT Artificial Intelligence Lab., AIM-433, July 1977.

Widlar Widlar, Robert J., "Design Techniques for Monolithic Operational Amplifiers" IEEE Journal Solid-State Circuits, vol. SC-4, pp. 184-191, August 1969.

Widlar Widlar, Robert J., "IC OP AMP Beats FETs on Input Current" EEE, December 1969.

Williams Williams, Brian C., "Qualitative Analysis of MOS Circuits" MIT Masters Thesis EECS January 1984

Appendix I - Phrase Grammar Rules

I.1 Summary of Types of Operational Amplifiers

The following kinds of circuits can be built with the given rules.

Standard Three stage operational amplifier with the following stages:

For the first stage:

Simple differential pair with resistive load
Simple differential pair with simple current mirror load
Simple differential pair with complex current mirror load
Darlington differential pair with resistive load
Darlington differential pair with simple current mirror load
Darlington differential pair with complex current mirror load
Super Beta differential pair with load
Current Cancellation Pair with load

For the second stage:

Simple common emitter
Darlington common emitter

For the third stage:

Standard push-pull output stage with two diode drop biasing
Standard push-pull output stage with 1 + 1/2 diode drop biasing

With these possibilities 32 different operational amplifiers can be synthesized.

1.2 Abstract Objects in the Phrase Grammar

1.2.1 Three Stage Operational Amplifier

;; this is the normal vanilla flavor op-amp, differential front end ;; simple amplifier for the gain stage, and buffer output stage. (to-make-an ?op-amp three-stage-operational-amplifier

```
   ;; Pattern of applicability:
   (where (type ?op-amp amplifier)
          (has ?op-amp (input differential))
          (has ?op-amp (output single-ended))
          (has ?op-amp (input voltage))
          (has ?op-amp (output voltage))
          (has ?op-amp (priority 1))
          (with-specs (gain-spec (voltage-gain ?op-amp))
                      (slew-rate-spec (slew-rate ?op-amp))
                      (input-offset-current-spec
                       (offset-current ?op-amp))
                      (input-bias-spec (bias-current ?op-amp))
                      (input-offset-voltage-spec (offset-voltage ?op-amp))
                      (drive-spec (output-drive ?op-amp))
                      (power-spec (power-consumption ?op-amp))))
   ;; Topology is as follows:
   ;; Parts are:
   (new-part (first-stage ?op-amp)
             ((type (first-stage ?op-amp) amplifier)
              (has (first-stage ?op-amp) (input differential))
              (has (first-stage ?op-amp) (output single-ended))
              (has (first-stage ?op-amp) (input voltage))
              (has (first-stage ?op-amp) (output current))
              (has (first-stage ?op-amp) (sign pnp)))
             ((use . ((gain gm) (slew-rate gm)))
              (with-specs
                (= (transconductance (first-stage ?op-amp))
                   (sqrt (* 2 (gain ?op-amp)))))))
   (new-part (second-stage ?op-amp)
             ((type (second-stage ?op-amp) amplifier)
              (has (second-stage ?op-amp) (input single-ended))
              (has (second-stage ?op-amp) (output single-ended))
              (has (second-stage ?op-amp) (input current))
              (has (second-stage ?op-amp) (output voltage))
              (has (second-stage ?op-amp) (sign npn)))
             ((with-specs
                (= (transresistance (second-stage ?op-amp))
                   (sqrt (* 2 (gain ?op-amp)))))))
   (new-part (feedback ?op-amp)
             ((type (feedback ?op-amp) capacitor)))
   (new-part (third-stage ?op-amp)
             ((type (third-stage ?op-amp) buffer)
              (has (third-stage ?op-amp) (input voltage))
              (has (third-stage ?op-amp) (output voltage))))

   ;; Connections to outside are:
   (connect (t+ ?op-amp) (t+ (first-stage ?op-amp)))
   (connect (t- ?op-amp) (t- (first-stage ?op-amp)))
   (connect (ot ?op-amp) (ot (third-stage ?op-amp)))
   ;; Internal connections are:
   (connect (ot (first-stage ?op-amp)) (it (second-stage ?op-amp))
            (t1 (feedback ?op-amp)))
   (connect (ot (second-stage ?op-amp)) (it (third-stage ?op-amp))
            (t2 (feedback ?op-amp)))
   ;; Propagating specifications to parts:
   ;; The gain of an op-amp is due to the product of the first two stages gain.
```

```
(= (gain ?op-amp)
   (* (transconductance (first-stage ?op-amp))
      (loading (second-stage ?op-amp) (first-stage ?op-amp))
      (transresistance (second-stage ?op-amp))))
(equation-with-variable-priority
 (= (transresistance (second-stage ?op-amp))
    (* (current-gain (second-stage ?op-amp))
       (input-impedance (third-stage ?op-amp))))
 (transresistance (second-stage ?op-amp)))
(equation-with-variable-priority
 (= (loading (second-stage ?op-amp) (first-stage ?op-amp))
    (// (output-resistance (first-stage ?op-amp))
        (+ (output-resistance (first-stage ?op-amp))
           (input-resistance (second-stage ?op-amp)))))
 (loading (second-stage ?op-amp) (first-stage ?op-amp)))
(equation-with-variable-priority
 (= (load-resistance (second-stage ?op-amp))
    (input-impedance (third-stage ?op-amp)))
 (load-resistance (second-stage ?op-amp)))
;; Power consumption is just the sum of the individual powers
(= (power-consumption ?op-amp)
   (+ (power-consumption (first-stage ?op-amp))
      (power-consumption (second-stage ?op-amp))
      (power-consumption (third-stage ?op-amp))))
;; simple equation for determining the slew rate.
(= (slew-rate ?op-amp)
   (// (max (current (ot (first-stage ?op-amp))))
       (capacitance (feedback ?op-amp))))
(= (cutoff-frequency ?op-amp)
   (// (transconductance (first-stage ?op-amp))
       (capacitance (feedback ?op-amp))))
;; The input offset current is one of the main specs given by the user.
(= (offset-current ?op-amp) (offset-current (first-stage ?op-amp)))
;; The input bias current is one of the main specs given by the user.
(= (bias-current ?op-amp) (* -1 (bias-current (first-stage ?op-amp))))
;; The input offset voltage is one of the main specs given by the user.
(= (offset-voltage ?op-amp) (offset-voltage (first-stage ?op-amp)))
;; The drive current is the amount of current that the amp can deliver to
;; a load of a specified size.  probably 2K ohms.
(= (drive-current ?op-amp)
   (maximum (current (ot (third-stage ?op-amp)))))
;; assumption is that most of the distortion comes from the third stage.
(= (distortion ?op-amp)
   (distortion (third-stage ?op-amp)))
;; might as well say what the third stage voltage gain should be.
(= (voltage-gain (third-stage ?op-amp)) 1))
```

1.2.2 Simple Differential Pair

```
(to-make-an ?dp simple-diff-pair
  ;; Pattern to match:
  (where (type ?dp amplifier)
         (has ?dp (input differential)) (has ?dp (output single-ended))
         (has ?dp (input voltage))      (has ?dp (output current))
         (has ?dp (sign ?sign))
         (has ?dp (priority 1))
         ;; something about a the sign
         (with-specs
           (= (transconductance ?dp)
              (* (bias-current ?dp)
                 (eval (if (eq (quote ?sign) 'NPN)
                           (get-algebra-value 'NPN-BETA)
```

```
                              (get-algebra-value 'PNP-BETA)))
                    q/kt))
              (= (offset-voltage ?dp) simple-offset-voltage)))
   ;; Parts are:
   (new-part (pos-q ?dp)
             ((type (pos-q ?dp) virtual-bjt-transistor)
              (has (pos-q ?dp) (sign ?sign)))
             ((use . ((gain gm)))))
   (new-part (neg-q ?dp)
             ((type (neg-q ?dp) virtual-bjt-transistor)
              (has (neg-q ?dp) (sign ?sign)))
             ((use . ((gain gm)))))
   (new-part (load ?dp)
             ((type (load ?dp) differential-pair-load)))
   (new-part (source ?dp)
             ((type (source ?dp) current-source)))

   ;; Connections to outside are:
   (connect (t+ ?dp)(base (pos-q ?dp)))
   (connect (t- ?dp)(base (neg-q ?dp)))
   (connect (emitter (pos-q ?dp))(emitter (neg-q ?dp))(t1 (source ?dp)))
   (connect (ot ?dp)(collector (pos-q ?dp)) (t2 (load ?dp)))
   (rail-connect (t2 (source ?dp))(rail (negative ?sign)))
   ;; Propagating specifications to parts:
   ;; To calculate bias current, assume that we can cheat and assume that
   ;; the bias current is basically one of the transistors input current.
   (= (bias-current ?dp) (current (base (pos-q ?dp))))
   ;; The offset current can be done by getting a delta-current from the
   ;; the load stage
   (= (offset-current ?dp)
      (// (delta-current (load ?dp))
          (beta (pos-q ?dp)))))
   ;; The voltage offset for a given differential pair can be calculated
   ;; once and then used as a constant.
   (= (offset-voltage ?dp) (offset-voltage (pos-q ?d)))
   ;; The transconductance is the gain element in the diff pair.
   ;; It's value depends on the type of load used.
   (= (transconductance ?dp)
      (* (gm (pos-q ?dp)) (transresistance-factor (load ?dp))))
   ;; so that the slew rate can be calculated it must know this
   (equation-with-variable-priority
    (= (max (current (ot ?dp)))
       (* -2 (current (collector (pos-q ?dp)))))
    (max (current (ot ?dp))))
   ;; output resistance
   (equation-with-variable-priority
    (= (output-resistance ?dp)
       (// (* (ro (pos-q ?dp)) (ro (t2 (load ?dp))))
           (+ (ro (pos-q ?dp)) (ro (t2 (load ?dp))))))
    (output-resistance ?dp))
   )
```

1.2.3 Simple Common Emitter

```
(to-make-an ?amp simple-common-emitter
   ;; Pattern to match:
   (where (type ?amp amplifier)
          (has ?amp (input single-ended)) (has ?amp (output single-ended))
          (has ?amp (input current))      (has ?amp (output voltage))
          (has ?amp (sign ?sign))
          (has ?amp (priority 1))
          (with-specs (transresistance-spec
                       (transresistance ?amp))
                      (power-consumption ?amp)))
```

```
   ;; Parts are
   (new-part (q ?amp)
             ((type (q ?amp) virtual-bjt-transistor)
              (has  (q ?amp) (sign ?sign)))
             ((use   ((gain beta)))))
   ;; Connections are:
   (connect (it ?amp) (base (q ?amp)))
   (connect (ot ?amp) (collector (q ?amp)))
   ;; some connection of emitter for biasing
   (connect (dc-bias ?amp) (emitter (q ?amp)))
   ;; Specifications
   (= (input-resistance ?amp)  (rpi (q ?amp)))
   (equation-with-variable-priority
    (= (max (current (ot ?amp))) (current (collector (q ?amp))))
    (max (current (ot ?amp))))
   (= (power-consumption ?amp)
      (* voltage-range (current (collector (q ?amp)))))
   (= (current-gain ?amp)
      (beta (q ?amp))))
```

1.2.4 Simple Common Collector

```
(to-make-an ?amp simple-common-collector
   ;; Pattern to match:
   (where (type ?amp common-collector)
          (has ?amp (input single-ended)) (has ?amp (output single-ended))
          (has ?amp (input voltage))      (has ?amp (output voltage))
          (has ?amp (sign ?sign))
          (has ?amp (priority 1)))
   ;; Parts are:
   (new-part (q ?amp)
             ((type (q ?amp) virtual-bjt-transistor)
              (has  (q ?amp) (sign ?sign))))
   ;; Connections are:
   (connect (it ?amp) (base (q ?amp)))
   (connect (ot ?amp) (emitter (q ?amp)))
   ;; some connection of emitter for biasing
   (connect (dc-bias ?amp) (collector (q ?amp)))
   (= (current-gain ?amp) (beta (q ?amp))))
```

1.2.5 Simple Follower

```
(to-make-an ?foll simple-follower
   ;; Pattern to match:
   (where (type ?foll follower)
          (has ?foll (sign ?sign))
          (has ?foll (priority 1)))
   ;; Parts
   (new-part (element ?foll)
             ((type (element ?foll) common-collector)
              (has (element ?foll) (sign ?sign))))
   ;; Connections
   (connect (ot ?foll) (ot (element ?foll)))
   (connect (it ?foll) (it (element ?foll)))
   (= (current-gain ?foll) (current-gain (element ?foll))))
```

I.2.6 Differential Pair Active Load

```
(to-make-an ?load differential-pair-active-load
   ;; Pattern to match:
   (where (type ?load differential-pair-load)
          (has ?load (sign ?sign))
          (has ?load (priority 1))
          (with-specs (delta-current-spec (delta-current ?load))
                      (transresistance-factor-spec
                       (transresistance-factor ?load))))
   ;; Parts are:
   (new-part (cm ?load)
             ((type (cm ?load) current-mirror)))
   ;; Connections are:
   (connect (t1 ?load) (reft (cm ?load)))
   (connect (t2 ?load) (outt (cm ?load)))

   (= (ro (t2 ?load)) (ro (outt (cm ?load))))
   (= (delta-current ?load) (delta-current (cm ?load)))
   (= 2 (transresistance-factor ?load)))
```

I.2.7 Differential Pair Resistive Load

```
(to-make-an ?load differential-pair-resistive-load
   (where (type ?load differential-pair-load)
          (has ?load (sign ?sign))
          (has ?load resistive)
          (with-specs (transresistance-factor-spec
                       (transresistance-factor ?load))
                      (delta-current-spec (delta-current ?load))))
   ;; Parts are:
   (new-part (r1 ?load) ((type (r1 ?load) resistor)))
   (new-part (r2 ?load) ((type (r2 ?load) resistor)))
   ;; Connections:
   (connect (t1 ?load)(t2 (r1 ?load)))
   (connect (t2 ?load)(t2 (r2 ?load)))
   (rail-connect (rail ?sign)(t1 (r1 ?load)))
   (rail-connect (rail ?sign)(t1 (r2 ?load)))
   ;; specifications
   ;; the delta current is dependent on how accurate we can make resistors
   ;; for a simple load
   (= (delta-current ?load) (* resistor-accuracy (current (t2 ?load))))
   (= (ro (t2 ?load)) (resistance (r2 ?load)))
   (= (ro (t1 ?load)) (resistance (r1 ?load)))
   (= 1 (transresistance-factor ?load))
   (= (resistance (r1 ?load))(resistance (r2 ?load))))
```

I.2.8 Simple Current Mirror

```
(to-make-an ?cm simple-current-mirror
  ;; Pattern to match:
  (where (type ?cm current-mirror)
         (has ?cm (priority 1))
         (has ?cm (sign ?sign))
         (with-specs (delta-current-spec (delta-current ?cm))))
  ;; Parts are:
  (new-part (diode ?cm)
            ((type (diode ?cm) virtual-bjt-transistor)
             (has (diode ?cm) (sign ?sign))))
```

```
  (new-part (output-trans ?cm)
            ((type (output-trans ?cm) virtual-bjt-transistor)
             (has (output-trans ?cm) (sign ?sign))))
  ;; Connect to output:
  ;; Also make diode into a diode
  (connect (refI ?cm)          (collector (diode ?cm))
                 (base (diode ?cm)) (base (output-trans ?cm)))
  (connect (outI ?cm) (collector (output-trans ?cm)))
  ;; Internal connections:
  (rail-connect (emitter (diode ?cm)) (rail (negative ?sign)))
  (rail-connect (emitter (output-trans ?cm)) (rail (negative ?sign)))

  ;Specifications
  (equation-with-variable-priority
   (= (ro (outI ?cm)) (ro (output-trans ?cm)))
   (ro (outI ?cm)))
  (= (delta-current ?cm) (// (current (collector (output-trans ?cm)))
                             (beta (output-trans ?cm))))
  (= (current (emitter (diode ?cm))) (current (emitter (output-trans ?cm)))))
```

1.2.9 Wilson Current Mirror

```
(to-make-an ?cm wilson-current-mirror
   ;; Pattern to Match:
   (where (type ?cm current-mirror)
          (has ?cm (sign ?sign)))
   ;; New Parts:
   (new-part (diode ?cm)
             ((type (diode ?cm) virtual-bjt-transistor)
              (has  (diode ?cm) (sign ?sign))))
   (new-part (output-trans ?cm)
             ((type (output-trans ?cm) virtual-bjt-transistor)
              (has  (output-trans ?cm) (sign ?sign))))
   (new-part (q ?cm)
             ((type (q ?cm) virtual-bjt-transistor)
              (has  (q ?cm) (sign ?sign))))
   ;; Connections:
   ;; connect to output
   (connect (refI ?cm) (collector (q ?cm)))
   (connect (refI ?cm) (base (output-trans ?cm)))
   (connect (outI ?cm) (collector (output-trans ?cm)))
   ;; internal connections
    ; make diode into a diode
   (connect (base (diode ?cm))(base (q ?cm))(collector (diode ?cm)))
   (rail-connect (emitter (diode ?cm))(rail ?sign))
   (rail-connect (emitter (q ?cm))(rail ?sign))
   (connect (emitter (output-trans ?cm))(collector (diode ?cm)))

   (= (ro (outI ?cm)) (ro (output-trans ?cm)))
   (= (delta-current ?cm) (// (current (collector (output-trans ?cm)))
                              (beta (output-trans ?cm))))
   (= (current (emitter (diode ?cm))) (current (emitter (output-trans ?cm)))))
```

I.2.10 Normal Op Amp Output Stage

```
(to-make-an ?buffer normal-op-amp-output-stage
   ;; Pattern to match:
   (where (type ?b buffer)
          (has  ?b (input voltage))
          (has  ?b (priority 1))
          (has  ?b (output voltage))
          (with-specs (power-consumption-spec (power-consumption ?buffer))))
   ;; Parts are:
   (new-part (input-stage ?b)
             ((type (input-stage ?b)
                    complement-pair-input-voltage-drop-input-stage)))
   (new-part (output-stage ?b)
             ((type (output-stage ?b) complement-pair)
              (has (output-stage ?b) (sign npn))))
   ;; Connections are:
   (connect (it ?b) (it (input-stage ?b)) (it2 (output-stage ?b)))
   (connect (ot ?b) (ot (output-stage ?b)))
   (connect (topt (input-stage ?b)) (it1 (output-stage ?b)))

   (equation-with-variable-priority
    (= (maximum (current (ot ?b)))
       (* -1 (current-gain (output-stage ?b))
          (current (it ?b))))
    (maximum (current (ot ?b))))
   (= (power-consumption ?b)
      (+ (power-consumption (input-stage ?b))
         (power-consumption (output-stage ?b))))
   ;; the distortion of this stage depends mostly on the type of
   ;; input stage used to bias the voltage drop across the output drivers.
   (= (distortion ?b) (distortion (input-stage ?b)))
   (= (input-impedance ?b)
      (* load-resistance (current-gain (output-stage ?b)))))
```

I.2.11 Complement Pair Input Voltage Drop Input Stage

```
(to-make-an ?cpvd complement-pair-input-voltage-drop-input-stage
   ;; Pattern to match:
   (where (type ?cpvd complement-pair-input-voltage-drop-input-stage)
          (has ?cpvd (priority 1))
          (with-specs (power-consumption-spec (power-consumption ?cpvd))))
   ;; New parts:
   (new-part (drop ?cpvd)
             ((type (drop ?cpvd) complement-pair-input-voltage-drop)))
   (new-part (load ?cpvd)
             ((type (load ?cpvd) resistor)))
   ;; Connections:
   (connect (it ?cpvd) (it (drop ?cpvd)))
   (connect (ot ?cpvd) (t1 (resistor ?cpvd)) (ot (drop ?cpvd)))
   (rail-connect (t2 (resistor ?cpvd)) (rail ?sign))

   (= (power-consumption ?cpvd)
      (* voltage-range (current (load ?cpvd))))
   (= (distortion ?cpvd) small-distortion)
   (= (bias-current (input-stage ?b))
      (current (t1 (resistor ?cpvd)))))
```

1.2.12 Simple Complementary Pair

```
(to-make-an ?cp simple-complementary-pair
   ;; Pattern to Match:
   (where (type ?cp complement-pair)
          (has ?cp (sign ?sign))
          (has ?cp (priority 1))
          (with-specs (power-consumption-spec (power-consumption ?cp))))
   ;; Parts are:
   (new-part (pull-up ?cp)
             ((type (pull-up ?cp) follower)
              (has (pull-up ?cp) (sign ?sign))))
   (new-part (pull-down ?cp)
             ((type (pull-down ?cp) follower)
              (has (pull-down ?cp) (sign (negative ?sign)))))
   ;; Connections are:
   (connect (ot ?cp) (ot (pull-down ?cp)) (ot (pull-up ?cp)))
   (connect (iu ?cp) (it (pull-up ?cp)))
   (connect (id ?cp) (it (pull-down ?cp)))

   (= (current-gain ?cp) (current-gain (pull-down ?cp)))
   (= (power-consumption ?cp)
      (* voltage-range (current (collector (pull-up ?cp))))))
```

1.2.13 Single Transistor Bjt

```
(to-make-an ?v-bjt Single-transistor-bjt
   ;; Pattern to Match:
   (where (type ?v-bjt virtual-bjt-transistor)
          (has ?v-bjt (sign ?sign))
          (has ?v-bjt (priority 1)))
   (new-part (q ?v-bjt)
             ((type (q ?v-bjt) bjt)
              (has  (q ?v-bjt) (sign ?sign))))
   (connect (base ?v-bjt)(base (q ?v-bjt)))
   (connect (collector ?v-bjt)(collector (q ?v-bjt)))
   (connect (emitter ?v-bjt)(emitter (q ?v-bjt)))
   (= (ro ?v-bjt) (ro (q ?v-bjt)))
   (= (rpi ?v-bjt) (rpi (q ?v-bjt)))
   (= (gm ?v-bjt) (gm (q ?v-bjt)))
   (= (beta ?v-bjt) (beta (q ?v-bjt))))
```

1.2.14 Double Darlington Transistor Bjt

```
(to-make-an ?v-bjt double-darlington-transistor-bjt
   ;; Pattern to Match:
   (where (type ?v-bjt virtual-bjt-transistor)
          (has ?v-bjt (sign ?sign))
          (has ?v-bjt (priority 2)))
   (new-part (q1 ?v-bjt)
             ((type (q1 ?v-bjt) bjt)
              (has (q1 ?v-bjt) (sign ?sign))))
   (new-part (q2 ?v-bjt)
             ((type (q2 ?v-bjt) bjt)
              (has (q2 ?v-bjt) (sign ?sign))))
   (connect (base ?v-bjt)(base (q1 ?v-bjt)))
   (connect (collector ?v-bjt)
                (collector (q1 ?v-bjt))
                (collector (q2 ?v-bjt))))
```

```
  (connect (emitter ?v-bjt)(emitter (q2 ?v-bjt)))
  (connect (emitter (q1 ?v-bjt)) (base (q2 ?v-bjt)))
  (= (rpi ?v-bjt) (* 2 (beta (q1 ?v-bjt)) (rpi (q2 ?v-bjt))))
  (= (ro ?v-bjt) (ro (q2 ?v-bjt)))
  (= (beta ?v-bjt) (* (beta (q1 ?v-bjt)) (beta (q2 ?v-bjt))))
  (= (gm ?v-bjt) (* (// 1 2) (gm (q2 ?v-bjt)))))
```

1.2.15 Super Beta Differential Pair

```
(to-make-an ?dp super-beta-diff-pair
  ;; Pattern to Match:
  (where (type ?dp amplifier)
         (has ?dp (input differential))
         (has ?dp (output single-ended))
         (has ?dp (input voltage))
         (has ?dp (output current))
         (has ?dp (sign npn))
         (has ?dp (priority 3))
         (with-specs (input-bias-spec (bias-current ?dp))
                     (offset-current-spec (offset-current ?dp))
                     (offset-voltage-spec (offset-voltage ?dp))
                     (gain-spec (transconductance ?dp))))

  ;; State the components:
  (new-part (pos-q ?dp)
            ((type (pos-q ?dp) super-beta-bjt-transistor)
             (has (pos-q ?dp) (sign ?sign))))
  (new-part (neg-q ?dp)
            ((type (neg-q ?dp) super-beta-bjt-transistor)
             (has (neg-q ?dp) (sign ?sign))))
  (new-part (pos-q2 ?dp)
            ((type (pos-q2 ?dp) virtual-bjt-transistor)
             (has (pos-q2 ?dp) (sign ?sign))))
  (new-part (neg-q2 ?dp)
            ((type (neg-q2 ?dp) virtual-bjt-transistor)
             (has (neg-q2 ?dp) (sign ?sign))))
  (new-part (diode1 ?dp)
            ((type (diode1 ?dp) diode)))
  (new-part (diode2 ?dp)
            ((type (diode2 ?dp) diode)))
  (new-part (diode3 ?dp)
            ((type (diode3 ?dp) diode)))
  (new-part (diode4 ?dp)
            ((type (diode4 ?dp) diode)))
  (new-part (source1 ?dp)
            ((type (source1 ?dp) current-source)))
  (new-part (source2 ?dp)
            ((type (source2 ?dp) current-source)))
  ;; Connect to outside terminals
  (connect (pos-it ?dp)(base (pos-q ?dp)))
  (connect (neg-it ?dp)(base (neg-q ?dp)))
  (connect (pos-ot ?dp)(collector (pos-q2 ?dp)))
  (connect (neg-ot ?dp)(collector (neg-q2 ?dp)))
  (connect (pos-it ?dp)(t1 (diode3 ?dp)))
  (connect (pos-it ?dp)(t2 (diode4 ?dp)))
  (connect (neg-it ?dp)(t2 (diode3 ?dp)))
  (connect (neg-it ?dp)(t1 (diode4 ?dp)))
  ;; now internal connections
  (connect (emitter (pos-q ?dp))(emitter (neg-q ?dp)))
  (connect (base (pos-q2 ?dp)) (base (neg-q2 ?dp))
           (t1 (diode1 ?dp)) (t2 (source2 ?dp)))
  (connect (collector (pos-q ?dp))(emitter (pos-q2 ?dp)))
  (connect (collector (neg-q ?dp))(emitter (neg-q2 ?dp)))
  (connect (t2 (diode1 ?dp))(t1 (diode2 ?dp)))
```

```
  (connect (t1 (source1 ?dp)) (t2 (diode2 ?dp)) (emitter (pos-q ?dp)))
  (rail-connect (t1 (source2 ?dp))(rail ?sign))
  (rail-connect (t2 (source1 ?dp))(rail (negative ?sign)))
  ;; Propagating specifications to parts:
  (= (offset-voltage ?dp) simple-offset-voltage)
  (= (transconductance ?dp)
     (* (gm (pos-q ?dp)) (transresistance-factor (load ?dp))))
  (= (bias-current ?dp)
     (* 2 (current (base (pos-q ?dp)))))
  (= (bias-current ?dp)
     (* 2 (current (base (neg-q ?dp))))))
```

1.7.16 Current Cancellation Differential Pair

```
(to-make-an ?dp current-cancellation-amp
  (where (type ?dp amplifier)
         (has ?dp (input differential))
         (has ?dp (output single-ended))
         (has ?dp (input voltage))
         (has ?dp (output current))
         (has ?dp (sign ?sign))
         (has ?dp (priority 2))
         (with-specs (input-bias-spec (bias-current ?dp))
                     (offset-current-spec (offset-current ?dp))
                     (offset-voltage-spec (offset-voltage ?dp))
                     (gain-spec (transconductance ?dp))))

  ;; State the components:
  (new-part (pos-q ?dp)
            ((type (pos-q ?dp) virtual-bjt-transistor)
             (has (pos-q ?dp) (sign ?sign))))
  (new-part (neg-q ?dp)
            ((type (neg-q ?dp) virtual-bjt-transistor)
             (has (neg-q ?dp) (sign ?sign))))
  (new-part (q3 ?dp)
            ((type (q3 ?dp) virtual-bjt-transistor)
             (has (q3 ?dp) (sign ?sign))))
  (new-part (q4 ?dp)
            ((type (q4 ?dp) virtual-bjt-transistor)
             (has (q4 ?dp) (sign ?sign))))
  (new-part (q5 ?dp)
            ((type (q5 ?dp) virtual-bjt-transistor)
             (has (q5 ?dp) (sign (negative ?sign)))))
  (new-part (q6 ?dp)
            ((type (q6 ?dp) virtual-bjt-transistor)
             (has (q6 ?dp) (sign (negative ?sign)))))
  (new-part (q7 ?dp)
            ((type (q7 ?dp) virtual-bjt-transistor)
             (has (q7 ?dp) (sign (negative ?sign)))))
  (new-part (q8 ?dp)
            ((type (q8 ?dp) virtual-bjt-transistor)
             (has (q8 ?dp) (sign (negative ?sign)))))
  (new-part (source1 ?dp)
            ((type (source1 ?dp) current-source)))
  (new-part (source2 ?dp)
            ((type (source2 ?dp) current-source)))
  (new-part (bias-q ?dp)
            ((type (bias-q ?dp) virtual-bjt-transistor)
             (has (bias-q ?dp) (negative ?sign))))
  (new-part (d1 ?dp) ((type (d1 ?dp) diode)))
  (new-part (d2 ?dp) ((type (d2 ?dp) diode)))
  (new-part (load ?dp)
            ((type (load ?dp) differential-pair-load)))
             ;(has  (load ?dp) resistive)
```

```
;; Now connect them to the outside
(connect (pos-it ?dp)(base (pos-q ?dp)) (collector (q6 ?dp)))
(connect (neg-it ?dp)(base (neg-q ?dp)) (collector (q8 ?dp)))
(connect (ot ?dp)(collector (q3 ?dp)) (t2 (load ?dp)))
(connect (neg-ot ?dp)(collector (q4 ?dp)) (t1 (load ?dp)))

;; now the inside connections on positive side
(connect (collector (pos-q ?dp))(emitter (q3 ?dp)))
(connect (emitter (pos-q ?dp))  (t1 (source1))
         (emitter (neg-q ?dp))  (base (bias-q ?dp)))
(connect (base (q3 ?dp))        (collector (q5 ?dp))
         (base (q5 ?dp))        (base (q6 ?dp)))
(connect (emitter (q5 ?dp))     (emitter (q6 ?dp))
         (emitter (q7 ?dp))     (emitter (q8 ?dp))
         (t2 (source2 ?dp))     (t1 (d1 ?dp)))

;; now the inside connections on neg side
(connect (collector (neg-q ?dp))(emitter (q4 ?dp)))
(connect (base (q4 ?dp))        (collector (q7 ?dp))
         (base (q7 ?dp))        (base (q8 ?dp)))

;; bias connections
(rail-connect (t1 (source2 ?dp))(rail ?sign))
(rail-connect (t2 (source1 ?dp))(rail (negative ?sign)))
(connect (t2 (d1 ?dp))(t1 (d2 ?dp)))
(connect (t2 (d2 ?dp))(emitter (bias-q ?dp)))
(rail-connect (collector (bias-q ?dp))(rail (negative ?sign)))

;; Propagating specifications to parts:
(= (current (emitter (q5 ?dp))) (current (emitter (q6 ?dp))))
(= (current (emitter (q7 ?dp))) (current (emitter (q8 ?dp))))
(= (offset-voltage ?dp) simple-offset-voltage)
(= (transconductance ?dp)
   (* (gm (pos-q ?dp)) (transresistance-factor (load ?dp))))
(= (bias-current ?dp)
   (+ (current (base (pos-q ?dp))) (current (collector (q6 ?dp)))))
;; The offset current can be done by getting a delta-current from the
;; the load stage
(= (offset-current ?dp)
   (// (delta-current (load ?dp))
       (beta (pos-q ?dp))))
;; so that the slew rate can be calculated it must know this
(equation-with-variable-priority
 (= (max (current (ot ?dp)))
    (* -2 (current (collector (pos-q ?dp)))))
 (max (current (ot ?dp))))
;; output resistance
(equation-with-variable-priority
 (= (output-resistance ?dp)
    (// (* (ro (pos-q ?dp)) (ro (t2 (load ?dp))))
        (+ (ro (pos-q ?dp)) (ro (t2 (load ?dp))))))
 (output-resistance ?dp)))
```

I.2.17 Cp Voltage Drop With Diode

```
(to-make-an ?vd cp-voltage-drop-with-diode
   ;; Pattern
   (where (type ?vd complement-pair-input-voltage-drop)
          (has ?vd (priority 1))
          (has ?vd (sign ?sign)))
   ;; Parts:
   (new-part (d1 ?vd)
             ((type (d1 ?vd) diode)
              (has (d1 ?vd) (sign ?sign)))))
```

```
   (new-part (d2 ?vd)
             ((type (d2 ?vd) diode)
              (has (d2 ?vd) (sign ?sign))))
   ;; Connections:
   (connect (it ?vd) (it (d1 ?vd)))
   (connect (ot ?vd) (ot (d2 ?vd)))
   (connect (ot (d1 ?vd)) (it (d2 ?vd))))
```

1.2.18 NPN Transistor Diode

```
(to-make-a ?d npn-trans-diode
   ;; Pattern
   (where (type ?d diode)
          (has ?d (priority 1))
          (has ?d (sign npn)))
   ;; New parts:
   (new-part (q ?d)
             ((type (q ?d) virtual-bjt-transistor)
              (has (q ?d) (sign npn))))
   ;; Connections:
   (connect (it ?d) (base (q ?d)) (collector (q ?d)))
   (connect (ot ?d) (emitter (q ?d))))
```

1.2.19 PNP Transistor Diode

```
(to-make-an ?d pnp-trans-diode
   ;; Pattern
   (where (type ?d diode)
          (has ?d (priority 1))
          (has ?d (sign pnp)))
   ;; New parts:
   (new-part (q ?d)
             ((type (q ?d) virtual-bjt-transistor)
              (has (q ?d) (sign pnp))))
   ;; Connections:
   (connect (it ?d) (base (q ?d)) (collector (q ?d)))
   (connect (ot ?d) (emitter (q ?d))))
```

I.3 Primitive Objects in the Phrase Grammar

I.3.1 Simple NPN BJT

```
(to-make-an ?bjt simple-npn-bjt
   ;; Pattern to match:
   (where (type ?bjt bjt)
          (has ?bjt (sign npn))
          (terminal-device ?bjt)
          (device-parameter (beta ?bjt))
          (device-parameter (gm ?bjt))
          (has ?bjt (priority 1))
          (three-terminal-device (base ?bjt)(emitter ?bjt)(collector ?bjt)))
   (equation-with-variable-priority
    (= (current (collector ?bjt))
       (* (beta ?bjt) (current (base ?bjt))))
    (current (collector ?bjt)))
   (= (beta ?bjt) npn-beta)
   (= (gm ?bjt)
      (* q/kt (current (collector ?bjt))))
   (= (rpi ?bjt)
      (// (beta ?bjt) (* q/kt (current (collector ?bjt)))))
   (equation-with-variable-priority
    (= (ro ?bjt) (// 200. (current (collector ?bjt))))
    (ro ?bjt))
   (= 0 (+ (current (collector ?bjt)) (current (base ?bjt))
           (current (emitter ?bjt))))))
```

I.3.2 Simple PNP BJT

```
(to-make-an ?bjt simple-pnp-bjt
   ;; Pattern to match:
   (where (type ?bjt bjt)
          (has ?bjt (sign pnp))
          (terminal-device ?bjt)
          (device-parameter (beta ?bjt))
          (device-parameter (gm ?bjt))
          (has ?bjt (priority 1))
          (three-terminal-device (base ?bjt)(emitter ?bjt)(collector ?bjt)))
   (equation-with-variable-priority
    (= (current (collector ?bjt))
       (* (beta ?bjt) (current (base ?bjt))))
    (current (collector ?bjt)))
   (= (beta ?bjt) pnp-beta)
   (= (gm ?bjt)
      (* q/kt -1 (current (collector ?bjt))))
   (= (rpi ?bjt)
      (// (beta ?bjt) (* q/kt -1 (current (collector ?bjt)))))
   (equation-with-variable-priority
    (= (ro ?bjt) (// -80. (current (collector ?bjt))))    ; pnp-ro
    (ro ?bjt))
   (= 0 (+ (current (collector ?bjt)) (current (base ?bjt))
           (current (emitter ?bjt))))))
```

1.3.3 Super Beta NPN BJT

```
(to-make-an ?v-bjt super-beta-npn-bjt
   ;; Pattern to Match:
   (where (type ?bjt npn-bjt)
          (terminal-device ?bjt)
          (device-parameter (beta ?bjt))
          (device-parameter (gm ?bjt))
          (has ?bjt (priority 1))
          (three-terminal-device (base ?bjt)(emitter ?bjt)(collector ?bjt)))
   (= (current (collector ?bjt)) (* (beta ?bjt) (current (base ?bjt))))
   (= (beta ?bjt) super-npn-beta)
   (= (gm ?bjt) (* q/kt (current (collector ?bjt))))
   (= (rpi ?bjt) (// (beta ?bjt) (* q/kt (current (collector ?bjt)))))
   (= (ro ?bjt) super-npn-ro)
   (= 0 (+ (current (collector ?bjt)) (current (base ?bjt))
           (current (emitter ?bjt))))))
```

1.3.4 Standard Resistor

```
(to-make-an ?r standard-resistor
   ;; Pattern to match:
   (where (type ?r resistor)
          (terminal-device ?r)
          (device-parameter (resistance ?r))
          (has ?r (priority 1))
          (two-terminal-device (t1 ?r)(t2 ?r)))
   (= (voltage ?r
      (* (current (t1 ?r)) (resistance ?r)))
   (= 0 (+ (current (t1 ?r)) (current (t2 ?r))))))
```

1.3.5 Capacitor

```
(to-make-an ?c capacitor
   ;; Pattern to match:
   (where (type ?c capacitor)
          (terminal-device ?c)
          (device-parameter (capacitance ?c))
          (has ?c (priority 1))
          (two-terminal-device (t1 ?c)(t2 ?c)))
   (= 0 (current (t1 ?c)))
   (= 0 (current (t2 ?c)))))
```